NIST

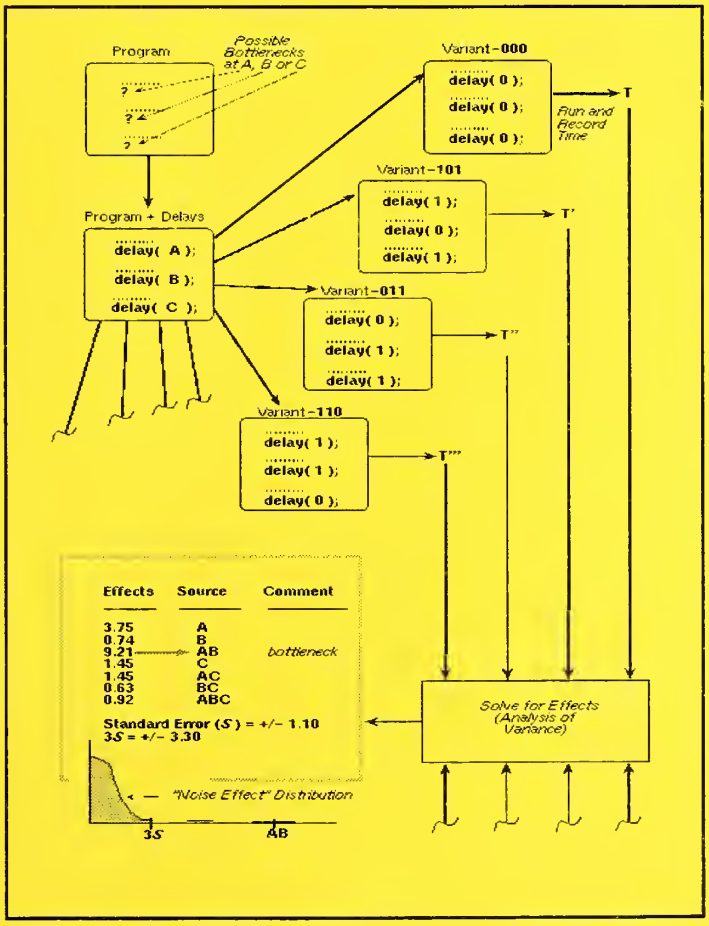
U.S. Department of Commerce
National Institute of Standards and Technology
High Performance Systems and Services Division
Scalable Parallel Systems and Applications Group

NISTIR 5789-1

Using S-Check ML

Version 3.0

Robert Snelick
Nathalie Drouin
John Antonishek
Mike Indovina
Michel Courson



February 1997
Supported by NIST task number 40131 and ARPA task number 7066.

NISTIR 5789-1

Using S-Check ML Version 3.0

Robert Snelick
Nathalie Drouin
John Antonishek
Mike Indovina
Michel Courson

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards
and Technology
High Performance Systems and
Services Division
Scalable Parallel Systems and
Applications Group
Gaithersburg, MD 20899-0001

February 1998

U.S. DEPARTMENT OF COMMERCE
William M. Daley, Secretary

TECHNOLOGY ADMINISTRATION
Gary R. Bachula, Acting Under Secretary
for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Raymond G. Kammer, Director

Preface

Today's multiprocessors provide unprecedented performance potential, yet all too often the actual performance obtained is far less impressive. The inherent complexity of parallel programs makes it far more difficult to capture true performance measurements on multiple-instruction stream, multiple-data stream (MIMD) architectures. In the absence of MIMD performance tools, obtaining reasonable parallel program performance is no small undertaking. The goal of S-Check is to provide a tool that gives the programmer useful performance information and is portable across machines as well as architectures.

S-Check automates the techniques of Synthetic Perturbation Screening (SPS). Synthetic Perturbation Screening systematically perturbs selected program code segments and determines performance sensitivities of these selected segments by using the statistical techniques of Design of Experiments (DEX). The resulting sensitivity analysis serves as a basis for performance evaluations. The name S-Check is derived from sensitivity analysis or **S**ensitivity **Check** (S-Check). S-Check ML is derived from S-Check's **M**ultiple **L**anguage (ML) capabilities. We will use the names S-Check and S-Check ML interchangeably throughout the text.

The concepts of Synthetic Perturbation for performance analysis of parallel programs were developed in the Parallel Processing Group at the National Institute of Standards and Technology (NIST) under the direction Dr. Gordon Lyon. In addition to Dr. Lyon and the authors, the following have contributed to the project: Dr. Raghu Kacker guided theoretical aspects of the statistical library developed for the tool; Dr. Joseph Ja'Ja' helped formulate some of the SPS techniques; Dominique Rodriguez wrote much of the tool's front end C parser and source code re-generator; Dr. James Filliben provided insight and examples for presenting statistical results graphically; Arnaud Linz provided assistance in testing the techniques and tool prototypes. Gordon Lyon also provided the diagram for the cover page. Ken Tice, Wayne Salamon, and Eric Lagergren read earlier versions of the text and suggested numerous details for improvement.

S-Check Version 3.0 is available for parallel SGI systems, IBM's SP machines, homogenous SUN, SGI, and RS6000 workstation clusters using

PVM or MPI, and PCs running Linux. S-Check provides support for using batch queuing systems (BQS) in homogeneous environments (e.g., IBM's SP2). S-Check provides interfaces for some of these BQS (e.g., LoadLeveler). If the BQS is not supported, the user can provide S-Check with linkage scripts required for integration. This scheme allows S-Check to run in many BQS environments. S-Check ML supports multiple languages, including FORTRAN 77, FORTRAN 90, C, and C++, however, advanced editing features are only available in C. Instrumentation code is provided for C and FORTRAN 90. For other languages (e.g., C++), the user must supply the instrumentation code. S-Check provides hooks for this purpose. The graphical interface is written with the widely available OSF Motif toolkit. When using this release of S-Check, it is important to remember that some planned features have not been completed or implemented at all; we point these out in the manual. This document supersedes NISTIR 5789 (February 1996), *Using S-Check, Alpha Release 1.0.*

Funding for the project is provided by NIST and the Advanced Research Projects Agency (ARPA), task number 7066. S-Check is public domain. All or any parts of it can be used, modified, or incorporated into other systems without permission from NIST or the authors. However, NIST and the authors would appreciate credit if the tool or parts of it are used. There is no warranty, expressed or implied, on the capabilities of the code. Certain commercial products are identified here in order to demonstrate use of the tool. Identification of such products does not imply recommendation or endorsement by NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

Send questions, comments, and bugs to **scheck-tool@www.scheck.nist.gov**. For information on how to obtain S-Check and for updates access the **http://www.scheck.nist.gov/scheck** Web site.

Robert Snelick
Nathalie Drouin
John Antonishek
Mike Indovina
Michel Courson

Gaithersburg, MD.
October 1997

Preface **iii**

INTRODUCTION *Using S-Check ML* ***1***

What is S-Check? **1**
S-Check ML Version 3.0 ***2***
Built-in Help ***3***

CHAPTER 1 *SPS Basics* ***5***

CHAPTER 2 *Getting Started* ***9***

Overview of an S-Check Experiment **9**
S-Check's Directory/Experiment Layout **11**
Creating/Opening an Experiment **12**

CHAPTER 3 *Experiment Configuration* ***15***

Configuring your Experiment **15**
Declaring the platform type **15**
Setting up LoadLeveler jobs. ***18***
Setting up POE jobs ***23***
Setting up PVM jobs ***24***
Setting up LAM MPI jobs. ***26***
Using the Generic platform. ***28***
Selecting your Language Type **28**
Connecting your application to S-Check **29**
Selecting the experiment type **30**

CHAPTER 4 *Instrumenting the Test Program* ***33***

Factor Editor **33**
Selecting Factors **35**
Instrumenting lines with preprocessor commands ***38***
Factor Profile and Automatic Factor Selection **38**

Factor Management **40**
Setting The Response Interval **40**
Instrumenting Programs Other Than C **43**
Instrumenting FORTRAN Programs ***43***

CHAPTER 5 *Experiment Control* ***45***

Experiment Control Window **45**
Experiment Maintenance and Convenience Functions ***46***
Launching Factor Editors ***46***
S-Check Messages ***46***
Setting the amount of delay ***48***
Selecting a DEX plan: ***49***
Setting Replication ***52***
Experiment Information Area ***53***
Running an Experiment **53**
Viewing Internal Results **55**
Obtaining an External Profile **57**

CHAPTER 6 *Viewing Performance Data* ***59***

List Effects **59**
Plot Effects **62**
Saving Results **65**
Viewing Multiple Displays **65**

CHAPTER 7 *Warnings and Bugs* ***67***

Warnings **67**
Bugs **69**

GLOSSARY *Glossary* ***71***

REFERENCES *References* ***73***

APPENDIX A *Error Messages* ***75***

APPENDIX B *Standard Error Table* ***79***

APPENDIX C *Code Instrumentation for the C Language* ***81***

APPENDIX D *Linking S-Check to Batch Queuing Systems* ***83***

INDEX *Index* ***91***

HOW TO INSTALL S-CHECK

S-CHECK QUICK-REFERENCE

INTRODUCTION

Using S-Check ML

Robert Snelick
Nathalie Drouin
John Antonishek
Mike Indovina
Michel Courson

What is S-Check?

S-Check is a software sensitivity checker designed to help you locate performance bottlenecks in parallel (and complex serial) programs. The tool S-Check provides the mechanisms to:

- determine the impact of computational code segments
- determine the cost of synchronization barriers
- detect interdependencies amongst code segments
- determine how well a program or code segment scales (not yet implemented)

S-Check employs and automates statistically designed experiments to identify sources of performance degradation. The host system can be a serial, parallel or networked (MPI/PVM-like) layout. The tool implements the techniques of Synthetic Perturbation Screening (SPS) developed at the National Institute of Standards and Technology (NIST). The methodology demands laborious experiment setup and execution procedures. S-Check automates much of SPS drudgery via an easy-to-use graphical user interface.

S-Check provides easy selection of test parameters, code instrumentation, experiment plan setup, experiment execution, calculation of results, and graphical presentation of results. S-Check is designed to accommodate users with varied SPS and design of experiment (DEX) skills. SPS proficient users manually control experiment details while novice users are given mechanisms for automatic setup and testing.

SPS introduces the notion of inserting artificial delays into the source code and capturing the effects of such delays by employing design of experiment techniques. Performance information takes the form of effects that correspond to source code segments or interactions among code segments. Effects are ranked by magnitude. Source code segments with the highest effects are likely candidates for bottlenecks. Based on variations of these techniques other performance information specific to a given architecture can also be obtained.

This document is provided as a user's guide to S-Check; it is not intended to describe or explain SPS techniques. However, a brief superficial overview of the process is given in the next chapter, *SPS Basics*. A detailed description of SPS can be found in the following publications: "Synthetic-perturbation tuning of MIMD programs" [1], "Synthetic perturbation techniques for screening shared-memory programs" [2], and "A simple scalability test for parallel code" [6]. An overview of the tool with case studies can be found in "S-Check: A Tool for Tuning Parallel Programs" [3] or "Tuning Parallel and Networked Programs with S-Check" [4].

For a Quick Reference Guide on S-Check usage look on the back of this manual. To see an example of an S-Check session, see "S-Check, by Example" [5]. Here the user is guided through a simple step-by-step example using S-Check. This is a good starting point to familiarize yourself to the basic S-Check concept. The user's guide can then be used more effectively as a reference guide and for using advanced S-Check features.

S-Check ML Version 3.0

S-Check ML Version 3.0 is available for parallel SGI systems, IBM's SP machines, homogenous SUN, SGI, and RS6000 workstation clusters using PVM or MPI, and PCs running Linux. S-Check provides support for using

batch queuing systems (BQS) in homogeneous environments (e.g., IBM's SP2). S-Check provides interfaces for some of these BQS (e.g., LoadLeveler). If the BQS is not supported, the user can provide S-Check with linkage scripts required for integration. This scheme allows S-Check to run in many BQS environments. Details of how to integrate S-Check and a BQS is described in Appendix D.

S-Check ML supports multiple languages, including FORTRAN 77, FORTRAN 90, C, and C++. However, advanced editing features are only available in C. S-Check versions prior to 3.0 only support the C language. Instrumentation code is provided for C and FORTRAN 90. For other languages (e.g., C++), the user must supply the instrumentation code. S-Check provides hooks for this purpose. The graphical interface is written with the widely available OSF Motif toolkit.

The name S-Check is derived from sensitivity analysis or **S**ensitivity **Check** (S-Check). S-Check ML is derived from S-Check's **M**ultiple **L**anguage (ML) capabilities. We will use the names S-Check and S-Check ML interchangeably throughout the text.

In this release, the framework for some features exists in the interface but have not yet been implemented. These features are *grayed out*--they are inaccessible. For example, scaling tests are not yet implemented. Some features that are not available in this release are nonetheless described.

Built-in Help

Built-in help is available while using S-Check. On various windows, access to the help service is provided with Help buttons. Click on the button to start the Help service. A list of topics are displayed. Click on the appropriate topic for help information.

CHAPTER 1

SPS Basics

S-Check is based on the techniques of Synthetic-Perturbation Screening (SPS) developed by Lyon, et al [1, 2]. A brief summary of that work is presented here. A reader familiar with the technique can skip to the next chapter, *Getting Started.*

The SPS technique inserts artificial code (delays) within segments of a parallel program and evaluates the effect of the delays on performance. The delays simulate an adjustment in efficiency in these code segments. SPS assumes that if a program code segment is highly sensitive to slight perturbations, then comparable segment improvements will boost performance correspondingly. The process gives a sensitivity analysis from which program problem areas can be identified. The underlying SPS foundation is the statistical methods of design of experiments (DEX). Diagram 1 summarizes the SPS technique.

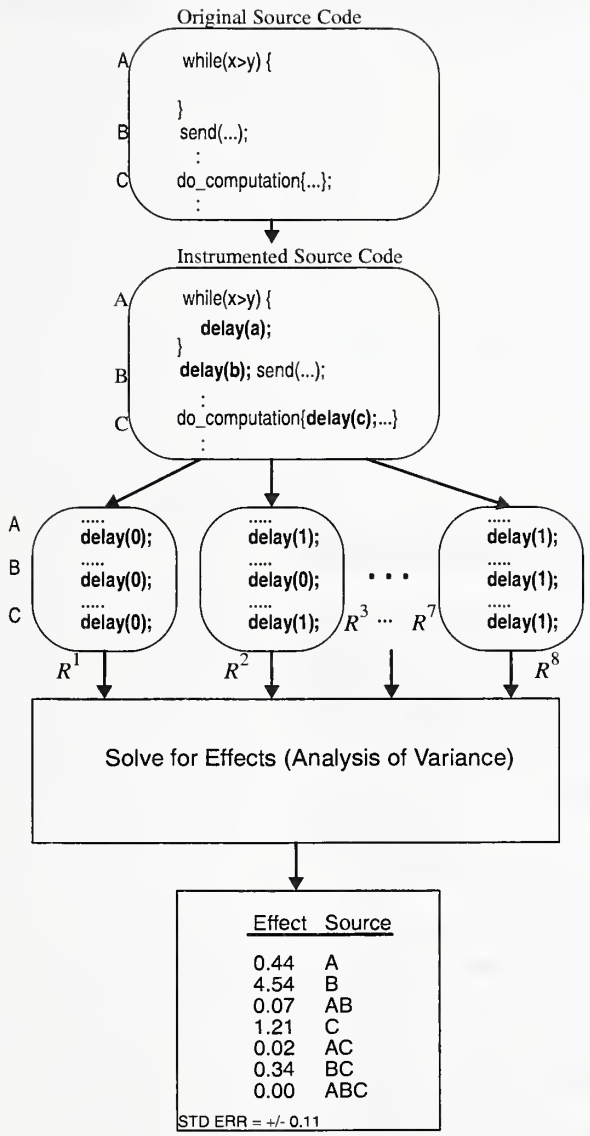
Within this framework, more specialized problems can be solved. These issues are quickly addressed below.

Screening test. At an early stage of analysis, one is primarily concerned with identifying and discarding factors that have no significant effect on the program response. Screening is an investigation strategy that efficiently iso-

lates important factors from a pool of candidates. Insignificant factors can then be removed from subsequent investigations, thus narrowing the scope of the analysis. Screening is an identification step that may be used to make a quick, preliminary assessment of a large application.

Barrier test. SPS can be used to address the issue of bottlenecks due to synchronization in the shared memory programming model. Processes that hit a barrier at widely dispersed times cause processors to idle for a significantly long period of time. The test objective is to identify barriers where such idle time overheads occur. Effects must be paired up and compared for analysis. In other words, two factors (or perturbations) are required per barrier. One is inserted immediately before the barrier, the other one immediately after. The barrier test requires a special factor treatment--not explained here. For each individual pair, the difference in the perturbation's respective effect gives an indication of the cost associated with each synchronization. The synchronization cost increases with the difference in the two effects. If the paired effects are about the same the synchronization cost is marginal. Note that screening experiments and barrier tests must be conducted separately as factor treatments in these two cases are not comparable. See the Reference [2], "Synthetic perturbation techniques for screening shared-memory programs" for a thorough explanation of the technique.

Scaling test. Code scalability determines how well parallel code avoids becoming a bottleneck as its host computer is made larger. Statistically designed experiments handle program and system together as a single entity. The system size (*i.e.,* the number of processors assigned to the program) comes as an additional factor to the regular set of input perturbations (*i.e.,* segments of code suspected of being performance bottlenecks). A large negative number for the effect associated with the system size means the whole program is sensitive to scalability, since the response time decreases with the system size. To determine which code segment improvements best promote parallel speedup the effects of interactions between code segments and the system size are studied. A large negative number for these interaction effects means the code seems to be scalable, since the sensitivity to delays decreases with the system size. See Reference [6] for details.



Step 1: Identify suspected bottleneck locations (called *factors*). For example, three factors are selected, a while loop, the send() function call, and the function do_computation{}. For illustrative purposes, we named the factors A, B, and C respectively.

Step 2: At each selected location, S-Check instruments the code with the capability to switch ON or OFF delays (ON = 1, OFF = 0). The delay simulates an adjustment in code efficiency.

Step 3: Guided by S-Check's DEX library, multiple versions of the program are generated, each with a different pattern of delays (DEX plan). As shown, the first version represents the original program since no delays (all 0) are activated. The next version, variant 101, will run with delay added to code segments represented by A and C; code segment B is undisturbed.

Step 4: Each version of the program is executed with delay settings and corresponding response time recorded. The response can be any useful measurement; typically the response is the total program execution time.

Step 5: The recorded response values ($R^1 - R^8$) and the DEX plan are used to solve for effects. In short, an effect of a factor is the difference between two average responses, one corresponding to executions when delay was ON for the factor, and the other, when the delay was OFF. An effect is a measure of how sensitive a code segment is to perturbation. The evaluations are quantitative and indicate the relative strength of any performance bottleneck.

Step 6: Examine and improve bottlenecks. Investigation (improvement) of code sections proceeds with the highest ranked effect in the list until a desired performance level is obtained. Our investigation would begin with the code segment represented by B (send()), since it is the most sensitive to perturbation. Alternatively, further exploration of the program can be performed by discarding some or all factors and adding more factors and then retesting. The process can be iterative.

Diagram 1. Summary of SPS Technique

CHAPTER 2

Getting Started

Overview of an S-Check Experiment

S-Check's basic notion is an *experiment*. An experiment defines all the parameters needed to setup and run the SPS process. S-Check views an experiment as an object that can be created, opened, initialized, saved, executed, displayed, and modified. Multiple experiments may be instantiated in an S-Check session. The following list provides an overview of the basic steps that need to be performed (from the user's perspective) in an S-Check experiment:

- Create/Open an experiment
- Declare the platform type
- Configure the test program
- Select the experiment type
- Select program test points
- Define a response interval
- Select an experimental plan
- Select experiment replication
- Run the experiment

- View experiment results

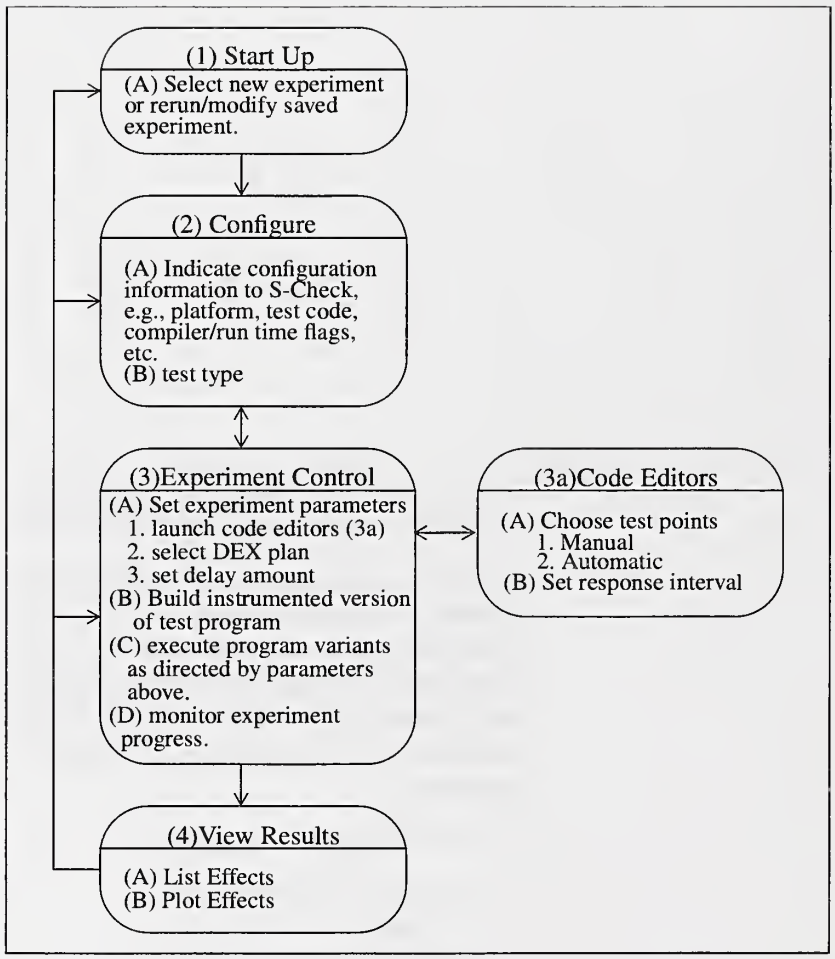


FIGURE 1. S-Check Usage Flow

S-Check assists in or performs each of these steps. Figure 1 shows a skeleton of S-Check's window interface layout. In the first two windows, Start-up (1) and Configure (2), system initializations and test code setup are performed. Experiment setup and control is handled in the Experiment Control

window (3). Test points are selected with Code (Factor) Editor windows (3a). The List and Plot windows (4) display results. The process can be repeated and entered at any step to refine the sensitivity analysis.

In addition to providing the necessary functions to run an experiment, S-Check organizes experiments by providing maintenance functions such as saving the experiment configuration, control settings, and results. S-Check also provides status information of the experiment. It indicates the number of test parameters selected, number of runs required to complete the experiment for a given plan type and replication setting, an estimate on how long an experiment will run, an estimate of the delay magnitude, state of an experiment, and trial number of a running experiment. Results from experiments take the form of rank ordered lists or graphical plots of effects. This data can be saved as postscript files.

S-Check's Directory/Experiment Layout

S-Check requires that all files needed to build the executable *test program* reside in the directory in which S-Check was started. As illustrated below in Figure 2, an experiment is saved in a sub-directory of the starting directory, called the *experiment directory.* The first level of this hierarchy is named ".scheck". Sub-directories with the name of the experiment are then created under the ".scheck" directory. These directories contain internal S-Check information for an experiment. To open a saved experiment, you must start S-Check in the appropriate experiment directory.

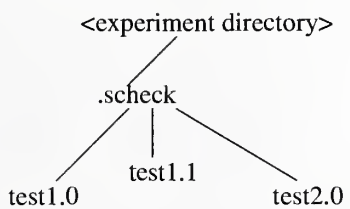


FIGURE 2. S-Check's directory hierarchy

To invoke S-Check, type the following command:

```
scheck
```

S-Check first displays a working dialog while it is performing system initializations. When this is completed the window for creating and opening experiments is launched.

Creating/Opening an Experiment

S-Check starts by displaying the Experiment List Window, as seen in Figure 3. The experiment directory name is displayed near the top of the Experiment List Window. S-Check will look here for *work set* files. The Experiment List Window displays previously defined experiments as well as an area for creating new experiments. To create a new experiment, enter the name of the experiment in the New Experiment Name field and select the *Open* button. To open a previously defined experiment double-click on the desired experiment in the experiment list.

Either of these actions brings up the Experiment Control Window (to be described shortly). This action also brings up the Configuration Window if the configuration for the experiment has not yet been defined. To delete an experiment, click on the experiment name and press *Delete*. To exit the Experiment List Window without requesting any tasks, select the *Cancel* button. To exit S-Check click on the *Quit S-Check* button. The *Cancel* button will also exit S-Check if no experiment is active.

Creating a new experiment from a previously defined experiment can really economize experiment setup time. Simply recall an experiment with a suitable configuration and use the *Save As* command on the Experiment Control Window menu. Modifications for the new experiment can now be made without having to re-initialize the bulk of experiment settings.

The Experiment List Window has two main menus: *Results* and *Info*. The menu selection *Multiple Display*, under *Results*, allows you to view results of previously saved experiments. The details of this are explained in *Viewing Results*. The *Info* menubar provides on-line information about S-Check.

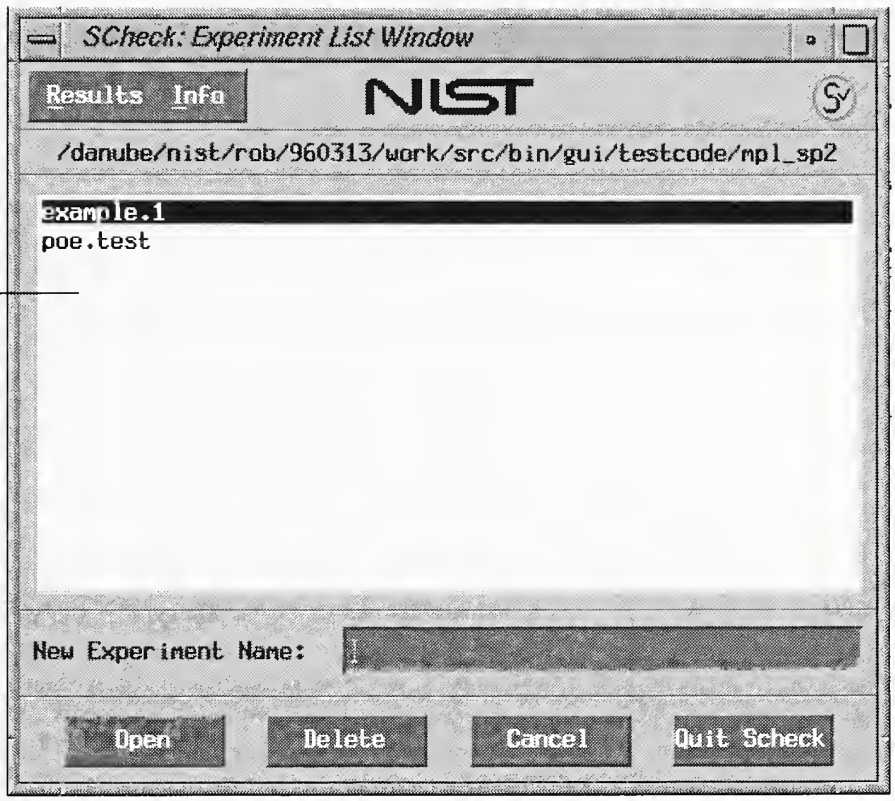


FIGURE 3. Experiment List Window

CHAPTER 3

Experiment Configuration

Configuring your Experiment

The Configuration Window (Figure 4) supplies information to S-Check about how to build the executable program, the language type, how and where to run the experiments, and the type of the experiment.

The Configuration Window is invoked automatically upon creation of an experiment. Alternatively, it can be explicitly brought up from the Experiment Control Window under the *File* and *Configure* menu selections.

Declaring the platform type

Use the platform type Option Menu to pick the platform you will run your experiments on. Specifying a platform type provides information to S-Check for both code instrumentation and about how to run your executable. S-Check supports a number of platforms. The UNIX selection is the default. Pick this type for running serial codes on workstations or parallel codes where executables run as simple Unix commands. The SGI Challenge is an

example of such a system. The platforms currently supported for the UNIX selection include Sun, SGI, and IBM RS6000 workstations and PCs running Linux.

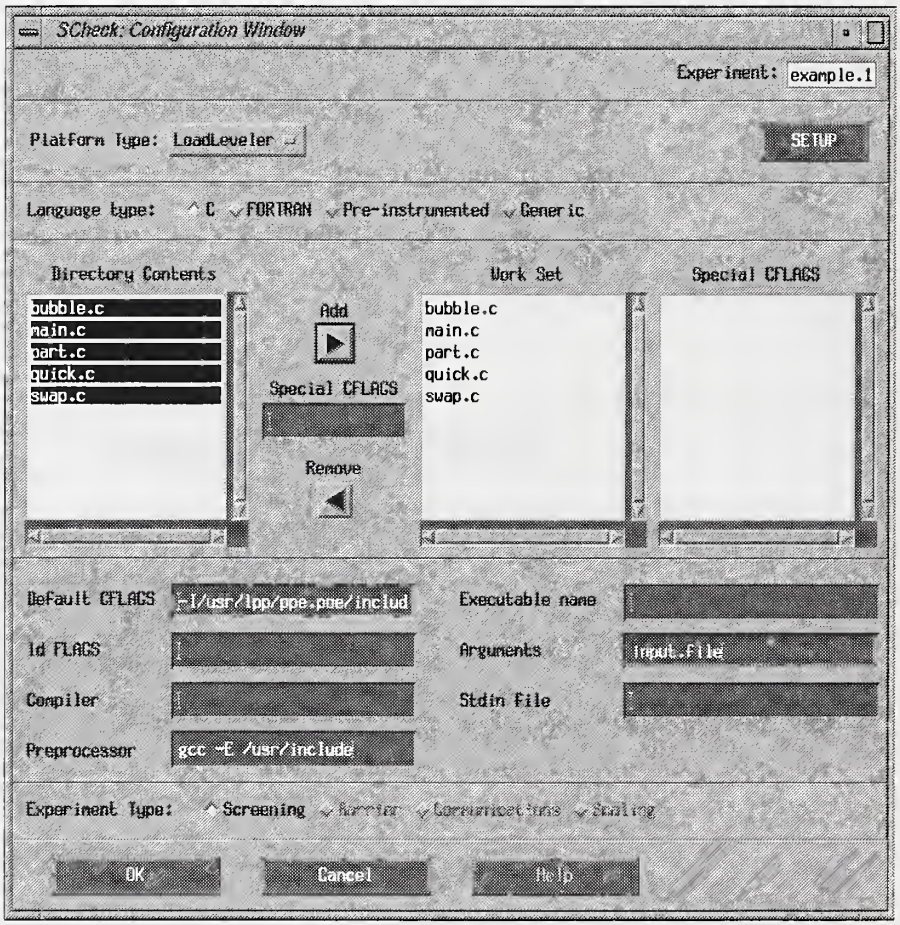


FIGURE 4. Configuration Window

When choosing the UNIX platform type, the user has the option of declaring a communication library. Click on the *Setup* button to the right of the platform *Option Menu* to bring up the Unix Setup Window (Figure 5). Here

a communication library can be selected. Declaring a communication library (*e.g.,* IRIX IPC) allows for automatic factor selection of the communication mechanisms. Automatic Factor Selection is discussed in Chapter 4. The *None* selection means that there is no default communication library. The *IRIX IPC* selection means that the IRIX IPC library calls can be automatically detected and instrumented. Modifications to the S-Check source code can be made easily to add additional communication libraries.

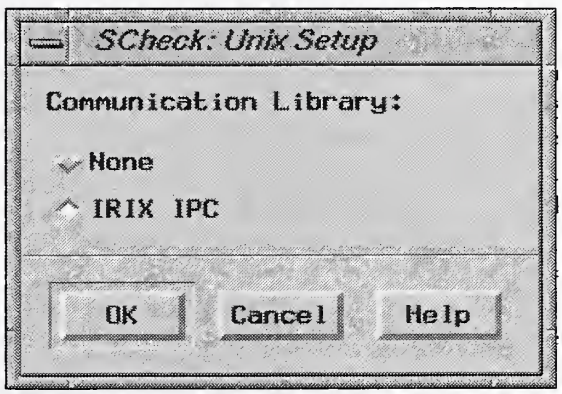


FIGURE 5. Unix Setup Window

S-Check can run jobs in other parallel and cluster environments as well. For these systems additional setup information is required. A dialog box will automatically appear upon selection of one of these platforms. The specific dialog box can be explicitly invoked (to change settings) by selecting the *Setup* button. In addition to the UNIX platform discussed above, S-Check supports POE, LoadLeveler, PVM, and LAM (MPI) jobs. The Generic platform is provided so that other scheduling systems can be linked to S-Check. However, the user (or system installer) must provide certain shell scripts necessary to link the systems (see Appendix D). Specific setup instructions for each of these platforms is given below.

Setting up LoadLeveler jobs.

Selecting a LoadLeveler job requires that you provide S-Check with a job command file. A job command file is a script that instructs Loadleveler how to schedule and run your job. The LoadLeveler Setup Window (Figure 6) will automatically appear upon selection of LoadLeveler platform. Enter the command file directly or select it using the file selection browser.

Running jobs with LoadLeveler may require that you provide additional information to S-Check in your job command file. You can run the job by providing a command file with a minimal set of LoadLeveler keywords or by modifying your existing command file by adding special S-Check tags. The former method combines your command file and S-Check default variables to define your job. The latter replaces special tags (that you define) with S-Check variables in your command file. These tags can define the test program, instrumentation environment variables, etc. The following examples illustrate how to setup SP LoadLeveler jobs using S-Check.

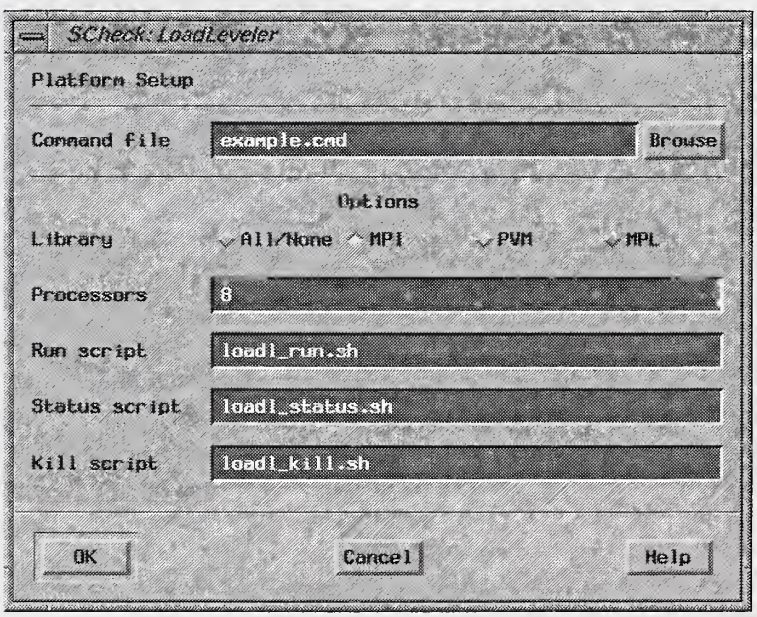


FIGURE 6. LoadLeveler Setup Window

The simplest way to connect your application to S-Check to run LoadLeveler jobs is to provide a job command file containing a minimal set of LoadLeveler keywords and let S-Check add its variables. Figure 7 shows an example of a simple job command file.

```
#!/bin/ksh
# ------------------------------------------------------------------
# simple.cmd: for loadleveler 1.2 05/30/96
# ------------------------------------------------------------------
#
# @ class               = Small
# @ min_processors      = 4
# @ max_processors      = 4
# @ job_type            = parallel
# @ queue
```

FIGURE 7. Simple Job Command File

Prior to execution, S-Check takes the command file and adds its variables necessary for defining the job. This includes statements which facilitate code instrumentation and program execution. The set of minimal keywords and keyword arguments needed to run a job may vary from system to system, you will need to determine these requirements for your particular setup. The commands in Figure 7 are sufficient for our testbed IBM SP2 system.

An alternative method for running LoadLeveler jobs is to use your existing command file and make modifications using S-Check variable tags. Figure 8 shows an example of a command file for running a job using LoadLeveler on an IBM SP2. Table 1 lists S-Check variables tags that can be used in addition to or to replace LoadLeveler keyword arguments and shell commands.

```
#!/bin/ksh
# ------------------------------------------------------------------
# run2.cmd: for loadleveler 1.2 05/15/96
# ------------------------------------------------------------------
#
# @ program              = qs
# @ initialdir           = /home/rob/quicksort
# @ output               = $(initialdir)/$(program).$(Cluster).out
# @ error                = $(initialdir)/$(program).$(Cluster).err
# @ requirements         = (Arch == "R6000") && (OpSys == "AIX41")
# @ class                = Small
# @ min_processors       = 4
# @ max_processors       = 4
# @ job_type             = parallel
# @ notification         = Never
# @ checkpoint           = no
# @ restart              = no

# @ queue
# initialization code (e.g., code to broadcast the data file in /tmp on ever y machine
# in the cluster before starting)

/usr/bin/poe /home/rob/quicksort/qs -p 4 -d 1048576

# clean up code
```

FIGURE 8. User's Original Job Command File

For example, wherever the LoadLeveler keyword *program* is defined and used, its argument needs to be replaced with the @SCHECK_program@ tag. Figure 9 shows how the original job command file in Figure 8 can be transformed so that S-Check experiments can be conducted.

In general, you substitute shell commands and LoadLeveler keyword arguments with corresponding S-Check variable tags. You build a job command file as before, but now S-Check tags are use in place of (and/or in addition to) certain variables. The first three S-Check tags in Table 1

(@SCHECK_initialdir@, @SCHECK_program@, @SCHECK_arguments@) correspond to LoadLeveler keywords. @SCHECK_trialnumber@ and @SCHECK_lasttrial@ provide information about the S-Check experiment. @SCHECK_delay@, @SCHECK_factors@, and @SCHECK_variables@ are used in instrumentation of the test program. @SCHECK_delay@ is the environment variable that contains the delay value as obtained from S-Check. @SCHECK_factors@ contain an encoded string that set S-Check factors ON or OFF. @SCHECK_variables@ tag sets and exports the environment variables previous discussed. This variable is explicitly added to the job command file if it is not found. If it is supplied by the user, it should proceed the test program invocation (*i.e.*, /usr/bin/poe ...).

Tag	Modification
@SCHECK_initialdir@	substitute for all instances of LoadLeveler keyword *initialdir.*
@SCHECK_program@	substitute for all instances of LoadLeveler keyword *program.*
@SCHECK_arguments@	substitute for all instances of LoadLeveler keyword *arguments*. Otherwise place tag on command line if user arguments are defined in the Configuration Window.
@SCHECK_trialnumber@	specifies the trial number of S-Check job, starts at 1.
@SCHECK_lasttrial@	indicates the last trial in S-Check job, set to 1 if last trial, 0 otherwise.
@SCHECK_delay@	S-Check environment variable SCHECKDELAY. Emits the delay value parameter set in S-Check.
@SCHECK_factors@	S-Check environment variable SCHECKFACTORS. Encoded variable for setting S-Check factors (ON or OFF).
@SCHECK_variables@	shell command that sets and exports S-Check environment variables.

TABLE 1. S-Check's tags for command files

```
#!/bin/ksh
# ------------------------------------------------------------------------
# run2.cmd: for Loadleveler 1.2, S-Check 1.0, 05/15/96
# ------------------------------------------------------------------------
#
# @ program              = @SCHECK_program@
# @ initialdir           = @SCHECK_initialdir@
# @ output               = $(initialdir)/$(program).@SCHECK_trialnumber@.out
# @ error                = $(initialdir)/$(program).@SCHECK_trialnumber@.err
# @ requirements         = (Arch == "R6000") && (OpSys == "AIX41")
# @ class                = Small
# @ min_processors       = 4
# @ max_processors       = 4
# @ job_type             = parallel
# @ notification         = Never
# @ checkpoint           = no
# @ restart              = no
# @ queue

@SCHECK_variables@
if [ @SCHECK_variables@ -eq 1 ] ; then
   # initialization code (e.g., code to broadcast the data file in /tmp on every machine
   # in the cluster before starting)
fi

/usr/bin/poe @SCHECK_initialdir@/@SCHECK_program@ @SCHECK_arguments@
if [ @SCHECK_lasttrial@ -eq 1 ] ; then
   # clean up code
fi
```

FIGURE 9. Modified Job Command File for S-Check

Prior to submission to LoadLeveler the *final* job command file is created. The tags are substituted as directed by the parameters set in S-Check. For example, S-Check will substitute the instrumented version (created by S-Check) of the test program at each @SCHECK_program@ tag. To inspect the job command file that is submitted to LoadLeveler, look at the

"scheck.*name*.cmd" file in the *experiment_directory*/.scheck/*experiment_name* directory.

Although tests have only been run with LoadLeveler on an IBM SP2 machine, it is possible to run LoadLeveler jobs with S-Check on other homogenous clusters. **However, testing of LoadLeveler outside the domain of an SP System has not been performed.**

Other LoadLeveler options in the LoadLeveler Setup Window that can be defined include the communication library, the number of processors, and the job control scripts. Selecting a communication library allows automatic selection and instrumentation of the library calls. The job control scripts (*i.e.*, run, status, and kill) control the LoadLeveler job. The default settings for these scripts are given on the LoadLeveler Setup Window. The user can modify these or create their own LoadLeveler scripts if desired; see Appendix D for details.

Setting up POE jobs

Selecting the SP2 POE platform also requires setup information. Upon selection of SP2 POE, the POE Arguments Window (Figure 10) will appear. Enter the argument value next to the argument qualifier for each argument you wish to set. See the example set up in Figure 10. Once you have set the arguments needed to run your executable click on the OK button. Note that at least one argument must be set, otherwise the platform type is reset to the default type (*i.e.*, UNIX). Selecting a message passing library allows automatic selection and instrumentation of the library calls.

The POE Arguments Window can be popped up at any time for modification of arguments by selecting the *Setup* button to the right of the platform *Option Menu*. For more information on POE arguments, see the *RS6000/SP Parallel Environment: Operation and Use* manuals [7].

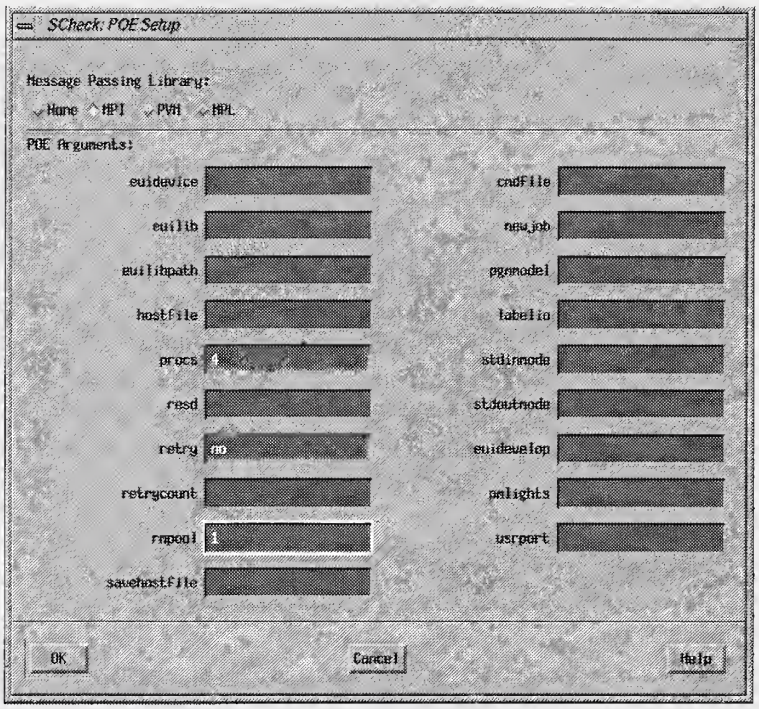


FIGURE 10. POE Arguments Window

Setting up PVM jobs

S-Check can be used to run PVM programs directly in a PVM environment such as a workstation cluster. However, currently support is limited to homogeneous clusters. PVM programs can be run in other environments as

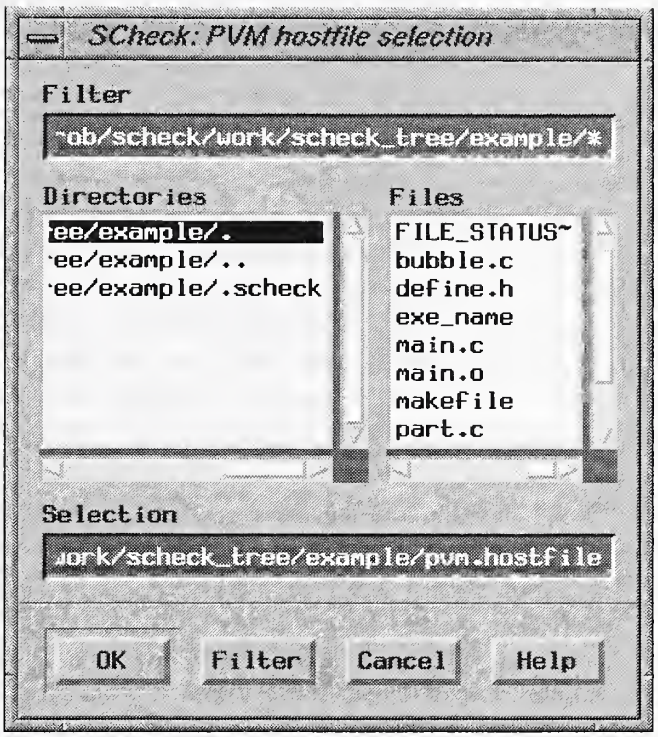


FIGURE 11. PVM Hostfile Selection Window

well, for example, LoadLeveler. To run PVM programs in environments other than a direct PVM environment, see the appropriate section in this chapter. This section describes how to run PVM jobs in a PVM environment.

To run a PVM job within S-Check you must first install and have PVM running. Configuring a PVM job proceeds much like a plain UNIX job described earlier, with a few exceptions. First, select PVM from the Platform Option menu. This brings up the PVM Hostfile Selection Window (Figure 11) requesting that you to select a PVM Hostfile defining the currently running PVM configuration. Select the hostfile and click *OK*.

S-Check makes certain assumptions about your PVM environment and imposes its own restrictions, these are outlined below:

- all hosts in the currently running PVM are homogeneous and are listed in a PVM "hostfile".
- S-Check is running on one of the hosts in the PVM (remote job execution is not yet available)
- each host in the PVM has enough disk space to store a copy of the instrumented executable, and a directory called $USER/pvm3/bin/$ARCH exists on each host which can be written to via "rcp" from the host currently running S-Check (*i.e.,* .rhosts permissions have been set up)
- the user has copied any data files which are needed on the remote nodes prior to running the experiment. Note that S-Check automatically copies the instrumented executable only to the remote nodes and not their associated data files.
- the application is SPMD and contains at least one call to *pvm_spawn()* in order to parallelize itself. The file(s) containing the *pvm_spawn()* call(s) have been loaded into the *work set* on the Configuration Window (described in the section, *Connecting your application to S-Check).* Note that the instrumented application will be given a different executable name than the one used by the *pvm_spawn()* in the un-instrumented code. S-Check will modify the *pvm_spawn()* to use the name of the instrumented executable. To disable this behavior for a specific *pvm_spawn()* call simply rename it to *scheck_pvm_spawn().*

In addition to these points, remember to add the "-I$*PVM_ROOT*/include" to the Default CFLAGS settings and the "-L/$*PVM_ROOT*/lib/$PVM_ARCH -lpvm3" to the ld FLAGS settings (where *PVM_ROOT* is the value of your PVM_ROOT environment variable) in the configuration text fields described in the section, *Connecting your application to S-Check.*

Setting up LAM MPI jobs.

S-Check can be used to run LAM programs directly in a LAM environment such as a workstation cluster. However, currently support is limited to homogeneous clusters only. When using S-Check to run LAM programs, S-Check makes certain assumptions about your environment:

- LAM is installed, and the virtual machine has been booted.
- All hosts in the currently running LAM are homogeneous and are listed in a LAM "boot schema."
- S-Check is running on the one of the hosts in the LAM virtual machine (remote job execution is not yet available)
- The file systems of each host are separate, *i.e.,* not shared (if any of the hosts share a filesystem it doesn't cause a problem, however S-Check will "rcp" the instrumented executable to each after the build).
- Each host in the LAM virtual machine has enough disk space to store a copy of the instrumented executable in the user's home directory on that system and the user's home directory is in their executable path. It is also assumed that the user has permission to "rcp" files to their home directory on each of the hosts in the LAM from the computer on which S-Check is being run.
- The user has copied any data files which are needed to the remote nodes prior to running the experiment. Note that S-Check automatically copies the instrumented executable only to the remote nodes and not their associated data files.
- The application is SPMD. Note that execution scripts (lam_run.sh, lam_status.sh, and lam_kill.sh) have been provided which assume the program can be run via the LAM utility **mpirun** without the use of an application schema file. Note, the instrumented application will be given a different executable name than the one you are currently using.
- The response interval has been placed around a sequential piece of code, which has access to the file system upon which S-Check is running (this practice should be followed for any application used with S-Check).

When configuring an experiment for use with LAM the following steps are necessary: (note, we assume your LAM software is installed in the directory /usr/local/lam60.)

- Set the Platform Type to "LAM"
- On the LAM Setup Window (Similar to Figure 6, LoadLeveler Setup Window) set the path to the LAM application schema (hostfile) which defines the currently running LAM configuration.
- Set the compiler to "hcc"

- Add "-I/usr/local/lam60/h" to the Default CFLAGS setting.

Using the Generic platform.

S-Check provides a method to link to other Batch Queuing Systems (BQS) provided that certain shell scripts can be written to give S-Check the capability to *run, kill*, and determine the *status* of a job. A description and example of how to link S-Check to a Batch Queuing System is described in Appendix D. This capability is a natural by-product from the integration of S-Check and LoadLeveler. The Generic hook is provided for users wanting to connect S-Check to non-supported BQS. The scripts must be provided at installation, prior to starting up S-Check. The setup for a Generic platform experiment can then proceed as a LoadLeveler setup does.

Selecting your Language Type

S-Check ML version 3.0 supports multiple languages, including C, C++, and FORTRAN. However, advanced editing features are only available in the C language. These features include error detection, automatic factor selection, and statement level instrumentation. Appendix C describes how the instrumentation process for the C language is implemented. Instrumentation for other languages is line oriented.

S-Check supports four language types: C, FORTRAN, Pre-instrumented, and Generic. Functionality for the language types differ significantly. The C language editors provides full editing capabilities. A scaled down line oriented editor is used for FORTRAN and other languages. It provides the basic functions needed to instrument S-Check experiments. S-Check Version 3.0 code instrumentation for FORTRAN is written in FORTRAN 90. Therefore, you must use a FORTRAN 90 compiler to use the built-in FORTRAN instrumentation. Your code, however, can be written in either FORTRAN 77 or FORTRAN 90. With the Pre-instrumented selection you can run programs that you have *pre-instrumented*. Pre-instrumenting a program entails entering the instrumentation code into your program with the use of an external editor such as *vi*. If you use a pre-instrumented program, you don't have to provide source file information to S-Check, only the name of

the executable (in the Executable Text Field). The Generic selection can be used for any language that you can provide instrumentation code for. S-Check provides the hooks to add instrumentation code so that most languages can be incorporated. The factor editor works the same as it does for FORTRAN. Sample instrumentation code for the C and FORTRAN 90 language can be found in the source code distribution. Details of factor selection and instrumentation can be found in Chapter 4.

Connecting your application to S-Check

The next step is to provide information so that S-Check can build your test program (see Figure 4). S-Check requires that you select files and provide flags that are needed to compile your program. S-Check also provides an area to declare command line arguments for the executable. The list, Directory Contents, names all allowable files from which an executable can be built. Select a file and enter any special C-flag(s) needed for that file. Push the *add* button to enter the file into the *work set*. The work set defines the files needed to build the executable. This list also defines the set of files that can be edited for choosing test locations (factors). The work set list appears on the Experiment Control Window. To remove a file from the work set, select the file and push the *delete* button.

Set default C-flags that apply to all C files in the Default CFLAGS text field (also use the C-flag field for other languages). For example, to instruct S-Check to look in the include directory named /usr/local/include, enter

```
-I/usr/local/include
```

in the Default CFLAGS text field.

If you are using special libraries for communication primitives you will need to indicate the location of the include files in the Default CFLAGS text field as well (e.g., /usr/lpp/ppe.poe/include). Likewise the loader/linker flags are set in the ld FLAGS text field. To specify a compiler, other than the default, enter the name of the compiler (path, if necessary) in the Compiler text field. Command line arguments for the test program are set in the area

named Arguments. Enter only the arguments. Do not enter the name of the executable.

To specify a preprocessor, other than the default, enter its name (path, if necessary) in the Preprocessor text field. Standard input files can be list in the Standard Input text field. List these as you would on a shell command but without the '<' metacharacter.

Special Note to AIX 4.1 users: It may be necessary to include the gnu C preprocessor in the Preprocessor text field in order for S-Check to parse your code. The preprocessor distributed with AIX provides a different format then the one used by S-Check. Until we correct this problem, use gcc -E -I/usr/include, in the Preprocessor text field. You will have to exit from S-Check and restart it for this change to take effect. Note that the gnu C compiler is required.

On our test system (IBM SP2 AIX 4.1) we set the Default CFLAGS text field to -I/usr/lpp/ppe.poe/include, the Compiler text field to /usr/bin/mpcc, and the Preprocessor text field to gcc -E -I/usr/include. On AIX 3.2, the default preprocessor is sufficient and the above changes are not required.

Selecting the experiment type

S-Check can perform basic screening tests to extract a sensitivity analysis of the test program (see bottom of Figure 4). Specialized tests are also available. For shared memory architectures, the cost of synchronization barriers can be evaluated. Communications test will restrict the test space to the communication library selected in one of the platform setup windows. Scaling is another test, one that determines how well a code segment scales as the size of the machine is increased.

The chosen experiment type dictates certain S-Check restrictions and requirements for additional input. For example, when performing barrier tests, only barriers can be selected as factors. Scaling tests require that the test program be capable of varying the number of processors. To select the experiment type, click on the desired test. Only one test can be selected.

Pressing *OK* will record this information and pop up the Experiment Control Window. *Cancel* will bring down the window with no changes or selections recorded.

CHAPTER 4

Instrumenting the Test Program

Factor Editor

Instrumenting the test program involves two functions:

- selecting factors
- defining the response interval

Factor selection involves picking code segments in the program that you wish to test. Wherever a factor is selected, S-Check inserts code that can be activated to cause a delay at that location. You can choose factors and the response interval by launching Factor Editors. To start a Factor Editor, double-click on the file you wish to edit in the Experiment Control Window (Figure 18 in Chapter 5). Starting a Factor Editor brings up a Factor Editor Window, loaded with the source code, and ready for factor selection. Figure 12 shows an example of a Factor Editor.

Some of the discussion on factor selection and defining the response interval applies only to the C language. There is a special section describing the differences between instrumenting C and FORTRAN programs. The differences in functionality are due to the fact that C code is parsed and regenerated, whereas simple text substitutions are made for instrumenting

FORTRAN and other languages. The instrumentation discussion will first be presented for the C language.

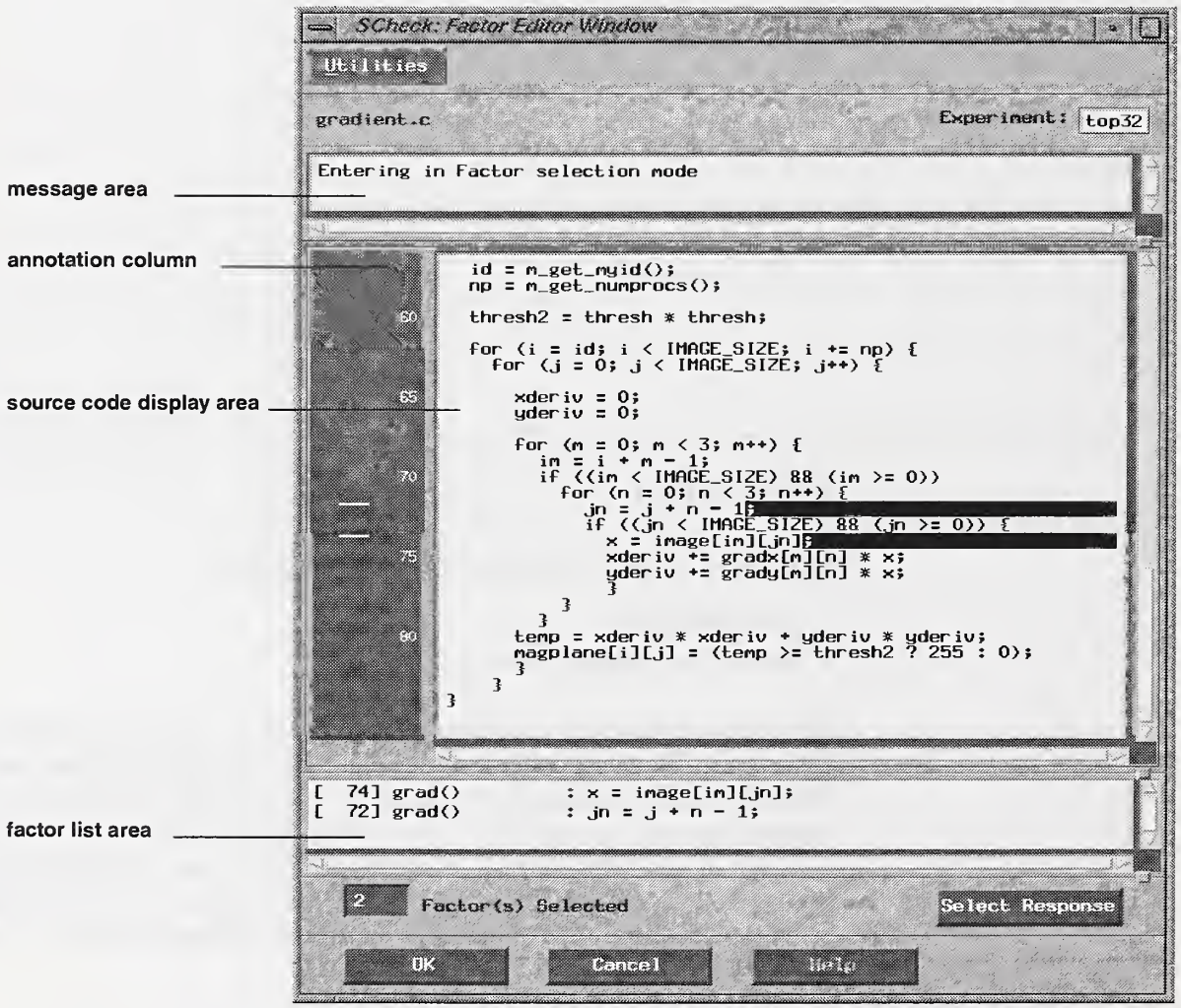


FIGURE 12. Factor Editor Window

Through the Factor Editor (Figure 12) you select factors and the response interval. The scrollable message area provides feedback during factor selection. For example, if you select an item that can not be instrumented, an appropriate message is displayed. The annotation column indicates if a factor is selected on a line by marking the corresponding annotation column line with a white horizontal dash. It also displays the start-stop indicators when the response interval is set. See discussion on *Setting the Response Interval*. The source code display area is where you define factors. Discussion on this topic is expanded in the next section, *Selecting Factors*. The factor list area provides a list of selected factors. The factors are displayed in sorted order depending on their line number within the file. The factor count displays the total number of factors selected in the file that is being edited. A count of all factors selected for the experiment is displayed on the Experiment Control Window. To the right of the factor count is the *Select Response* button. The purpose of this button is described in *Setting the Response Interval.*

Press *OK* to dismiss the Factor Editor and record all changes. To dismiss the window while ignoring modifications, press *Cancel.*

The size of the scrollable windows (*i.e.,* message area, source code display area, and the factor list area) can be increased/decreased by moving the panes at the lower right hand side of each scrollable window.

Selecting Factors

Factors are test points in the code. You are responsible for selecting factors. Factors can be selected manually (by clicking at a code location) or semi-automatically (by using S-Check's factor selection utility). The notion of a factor comes from the branch of statistics called design of experiments (DEX). In DEX, factors correspond to parameters that are varied in the experiment. To select a factor, click on the location where you want to define a test point. If this location is a valid factor that can be instrumented, the location of the instrumentation is indicated. The location is tagged with reverse video (Figure 12). The instrumentation will be inserted between the left most character and the right most character of the reverse video. Some

exceptions to this rule exist and are pointed out below. Click on the location again to remove the factor. The factor will no longer be highlighted in reverse video. Once a factor has been selected the factor count is updated for the current file that is being edited. The global factor count for the experiment is maintained on the Experiment Control Window.

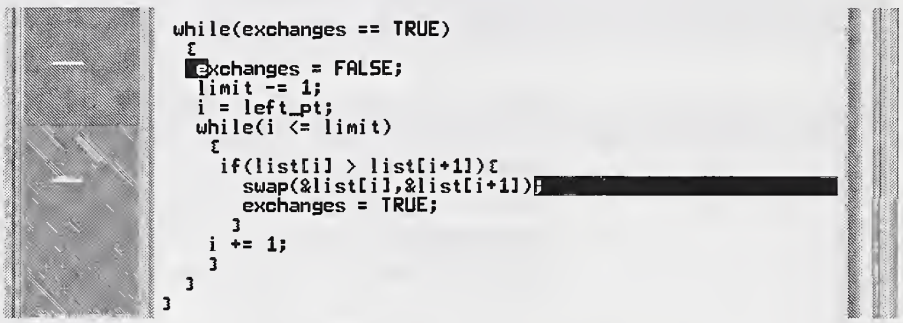


FIGURE 13. Factor Selection, Default Interpretation.

Figure 13 shows a cut out of a Factor Editor. The first selection instruments the body of the **while()** loop. That is, the perturbation code is inserted as the first statement in the loop. The second selection instruments the code after the call to **swap()** and before the statement **exchanges = TRUE**.

Defining a code segment to be a factor will cause instrumentation code to be inserted at that location. The code instrumentation process occurs later in the SPS process. In most cases the instrumentation code is inserted between the left most and right most character displayed in reverse video. In a few situations this rule does not apply. The user must exercise caution when selecting factors to insure that the intended location for instrumentation is the desired location. The example above shows the default interpretation.

The example in Figure 14 illustrates the special cases that apply to compound statements.

```
<F1> <D1> while <F2> (index > limit) <F3> { <F4>
              <D3> <F5> <D5>  stmt1 = setting1;
                              stmt2 = setting2;
              <D4>
              }
              <D2>
```

FIGURE 14. Factor Selection, Special Cases for Compound Statements.

The symbols <F1> through <F5> indicate the test locations you selected. In S-Check these locations will be highlighted in reverse video. The symbols <D1> through <D5> indicate the locations where the delay treatment is inserted into the code. Factors <F1> and <F5> adhere to the default interpretation as the delay is inserted where the factor was chosen. The interpretation for factor <F2> is to instrument the code following the while() loop (<D2>). Clicking at location <F3> instruments the body of the loop, that is, the first statement in the loop. It is instrumented exactly like <F5>. If the location F4 is selected, then the last statement in the while loop is instrumented (<D4>). The impact of <F4> and <F3/F5> will differ if there is branching back to the beginning of the loop. Some of these special case options are implemented for future S-Check features. If you want to instrument the body of the loop, it is best to use the default interpretation as in <F5>.

If you select a location that S-Check cannot instrument, you are notified of this in the message area. Items that cannot be instrumented include lines that are encased in preprocessor commands which are not defined and therefore will not be part of the executable code. Locations that produce invalid code (*e.g.,* at the end of a function) cannot be instrumented.

Instrumenting lines with preprocessor commands

Lines containing C preprocessor macro definitions can be expanded to show the real value of the macro definition. This aids in determining the actual location of the delay for statements containing these macros. Otherwise, the Factor Editor may misrepresent the location of the delay because it highlights the (displayed un-expanded) code according to the expanded or preprocessed code. S-Check provides a feature to toggle between the two states.

To expand a line that contains a macro, simply click on the line. All the instrumentation rules as described above apply as usual. To revert to the pre-expanded state, click on EXP in the annotation column.

Factor Profile and Automatic Factor Selection

S-Check provides two utilities to aid the process of factor selection. The Factor Profile Window (Figure 15) indicates the number of programming constructs or library (function) calls in the program or in a particular file. Based on this information the user can select factors automatically. An example selection command could be select all *for loops* in file compute.c.

To access this facility (at the program level) select *Factor Profile* button under the *Utilities* menu on the Experiment Control Window. The factor profile and automatic factor selection facility can also be accessed at the file level in each instance of a Factor Editor.

To obtain a Factor Profile, click on the desired construct button(s) and a listing showing the number of the selected construct(s) appears for each file defined in the work set. For example, to see how many for loops exist in the program, click on the for loops toggle button. This action will then indicate the number of for loops for each file in the program. A Factor Profile such as this can be obtained for: for, while, and do loops, function declarations, function calls, predefined function calls such as barriers() or calls to a communications library routine. Special functions calls can be added to S-Check default list (prior to installation) or declared explicitly in the Function Calls Text Field.

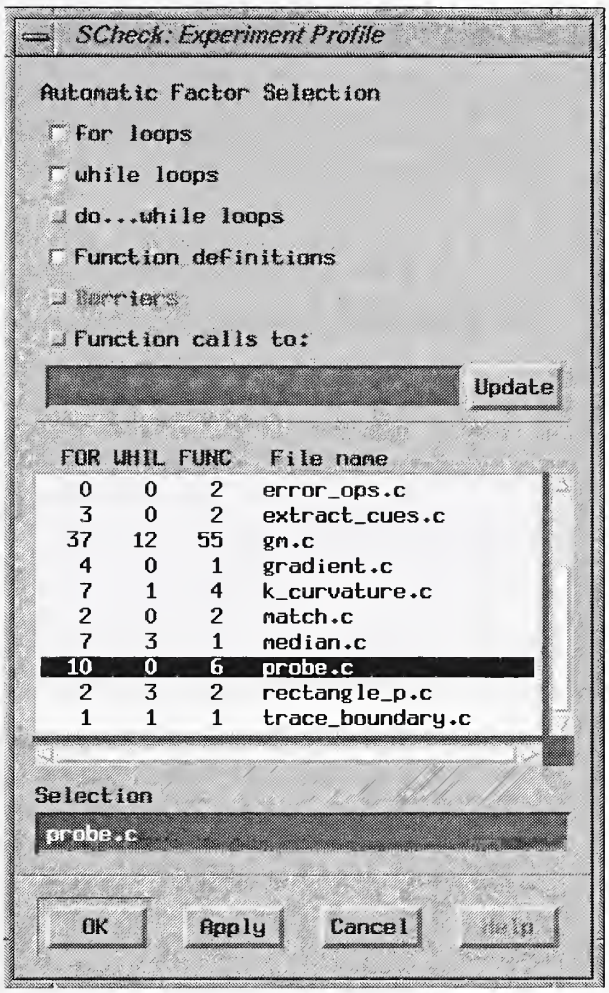


FIGURE 15. Factor Profile and Automatic Factor Selection

The information presented in a Factor Profile can aid in automatic factor selection. To instrument a group of factors in a file for a particular construct or constructs select the construct(s) and the file. Click on the *Apply* button to execute the request. For example, to instrument all for loops in the file gradi-

ent.c: (i) click in the for loops toggle button (ii) click on the file gradient.c (iii) click on the *Apply* button. Upon completion of the command, the factors just selected can be viewed with the appropriate Factor Editor instance. All factor counts will be updated automatically. See the Factor Management Window in the next section for group removal of factors.

Factor Management

The Factor Management Window (Figure 16) allows you to remove all factors in the experiment or all factors in an individual file. You access the feature by using the *Utilities* menu on the Experiment Control Window and selecting the *Factor Management* button. Click on "All files" and hit the *Apply* button to delete all factors in the experiment. To remove all factors in a particular file, click on that file and select the *Apply* button to carry out the instruction. To exit the window, select the *OK* button.

Setting The Response Interval

The second item that needs to be set for instrumenting the code is the response interval. The *response interval* defines the start and stop locations for capturing the response time. The response time equals the time recorded at the stop location minus the time recorded at the start location. The response time is used in the calculation of effects. The start/stop locations can be set in any file and the response interval can span files. That is, the start/stop locations need not reside in the same file. The response interval should be set so that it encompasses all factors. Otherwise, the effect for the factors not in the domain of the response interval is not accounted for. It is important that the response interval be set in a sequential part of the code or that it is guaranteed to be set by only one process. Results are otherwise undefined.

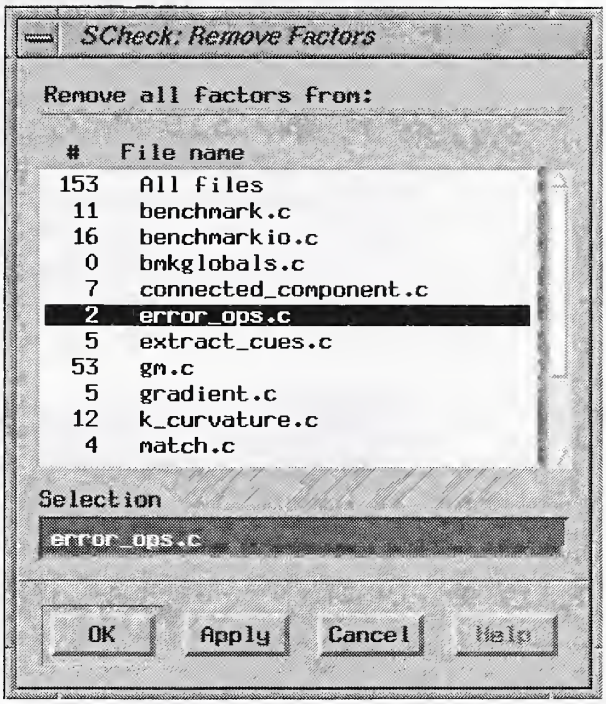
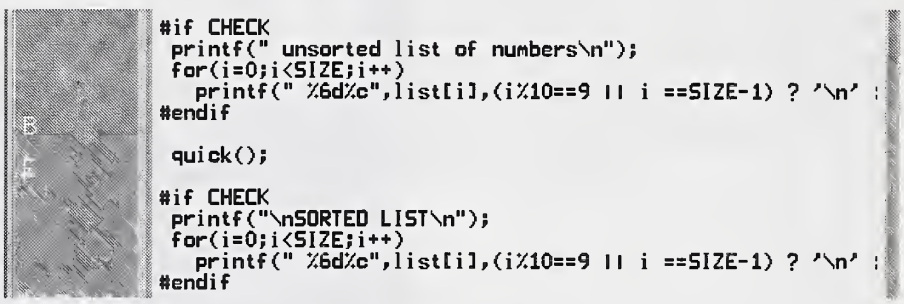


FIGURE 16. Removing Factors

To select the response interval click on the *Select Response* button (Factor Editor). This will turn off the factor selection mode and activate the select response mode. While in the select response mode, the cursor is modified to a down arrow indicator or to an up arrow indicator. Clicking in the source code display area while the down arrow cursor is active will cause a "B" to be placed in the annotation column corresponding to the selected location. Likewise, clicking in the source code display area while the up arrow cursor is active will cause an "E" to be placed in the annotation column corresponding to the selected location. The "B" indicates the start (begin) location and the "E" marks the stop (end) location.

The first time the *Select Response* button is clicked, the down arrow indicator is active. You can change the selection mode to an up arrow (for defining

the stop location) by selecting a start location or by clicking on the *Select Response* button. The latter option allows you to avoid setting or changing the start location. The former option will define the start location and automatically put the editor into the define stop location mode. Clicking twice on the *Select Response* button will return you to the factor selection mode (bypassing the stop location mode). The factor selection mode is automatically entered after the stop location is defined. Unlike factor selection, defining the response interval is line oriented. The resolution of the "B" and "E" (in this implementation) are somewhat coarse, so make sure the start and stop locations are clearly defined.



```
#if CHECK
 printf(" unsorted list of numbers\n");
 for(i=0;i<SIZE;i++)
   printf(" %6d%c",list[i],(i%10==9 || i ==SIZE-1) ? '\n' :
#endif

 quick();

#if CHECK
 printf("\nSORTED LIST\n");
 for(i=0;i<SIZE;i++)
   printf(" %6d%c",list[i],(i%10==9 || i ==SIZE-1) ? '\n' :
#endif
```

FIGURE 17. Defining the Response Interval.

In Figure 17, the function **quick()** is defined to be the response interval.

S-Check does not check for multiple execution of start/stop locations. It does however check if the stop location is executed before the start location. Only one response interval can be set. Setting the start/stop locations when the response interval is already defined will overwrite any previous settings. The new start/stop locations will define the response interval.

Setting the response interval will automatically reset the delay value to an undefined state. It also changes the job status to Empty. A *built experiment* will have to be reconstructed.

Instrumenting Programs Other Than C

The discussion above focused on instrumenting the test program for the C language. The instrumentation process for C differs from that of other languages supported by S-Check. This is due to the fact that C code is parsed and regenerated whereas the instrumentation for other languages (FORTRAN 90) is performed by text substitution. The basic instrumentation process still holds in both cases. However, some of the convenience functions provided for C are not available for other languages. These features include statement level instrumentation, verification of correct instrumentation placement, factor profile, and automatic factor selection.

Factor selection proceeds much the same way as described in the discussion above. Start a Factor Editor and click on locations where you want the instrumentation code. Instrumentation is line oriented. The delay code is placed on the next line. Make sure that the instrumentation code is placed at a location that still will produce correct (compilable) code.

Instrumenting FORTRAN Programs

Instrumentation of a FORTRAN program proceeds as described in the previous sections. FORTRAN editing is line oriented. S-Check Version 3.0 code instrumentation for FORTRAN is written in FORTRAN 90. Therefore you must use a FORTRAN 90 compiler to use the built-in FORTRAN instrumentation. Your code, however, can be written in either FORTRAN 77 or FORTRAN 90. If you do not have a FORTRAN 90 compiler, you can use the Generic Language Type setting on the Configuration Window and supply your own instrumentation code that works with your particular compiler.

CHAPTER 5

Experiment Control

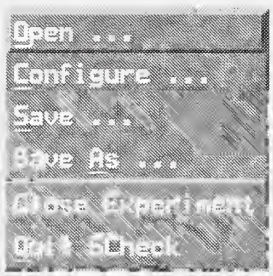
Experiment Control Window

The Experiment Control Window (Figure 18) involves:

- Experiment maintenance and convenience functions (*File* menubar)
- Launching factor editors
- Setting and showing the delay value (*Utilities* menubar)
- Selecting a DEX plan
- Selecting experiment replication
- Displaying progress and error messages from S-Check
- Displaying experiment setup information
- Running an experiment
- Displaying run status information
- Viewing internal results
- Obtaining an external profile
- Saving Results
- Launching result viewers (*Display* menubar)

The first seven topics are described in this section. The next two are explained in the section to follow, *Running an Experiment*. Viewing internal results and obtaining an external profile are explained in separate sections. The last two items are covered in *Saving Results* and *Viewing Results*.

Experiment Maintenance and Convenience Functions



Experiment maintenance and convenience functions are accessed under the *File* menubar selection. *Open* gives you access to the Experiment List Window. For experiment configuration modifications select the *Configure* button to bring up the Configuration Window. To save the current experiment, select *Save*. This action brings up a box with toggle buttons named *Save Experiment* and *Save Results*. To save all of the settings of an experiment, select *Save Experiment* and click-on *Save*. The saved file contains everything necessary to retrieve an experiment in its present state. These items include everything defined on the Configuration Window, Experiment Control Window and Factor Editors. It also saves the state of the interface. A saved experiment can be opened from the Experiment List Window. Results of an experiment can be saved by selecting the *Save Results* toggle button and pressing on *Save*. This action pops up the Save Results Window. See the section on *Saving Results* for details. To save an experiment with a new name, select *Save As*. To close an experiment, select *Close*. *Quit SCheck* will exit the program.

Launching Factor Editors

The work set is the list of files defining the target program. This list is constructed in the Configuration Window. Double-click on the file you wish to edit. This action launches a factor editor, loaded with source code, and ready for factor selection. There is no limit on the number of factor editors you can launch. See *Instrumenting the Test Program* for details on using a factor editor.

S-Check Messages

S-Check uses the message area to inform you about its progress in instrumenting the target program. Typical status information includes:

- files being parsed

- compilation progress
- stages in determining the delay value
- state and value of the delay
- traces of response time and treatments for trials
- information for effect calculations
- warnings and errors

TABLE 2. Experiment Control Warnings and Error Messages

Message	Reason	Fix/Consequent
delay value is no longer set	1. factor set was changed. 2. response interval was changed.	1. a built experiment has to be reconstructed because the previous delay value may not be appropriate for the new experiment settings.
unable to find suitable delay value in "*x*" tries	1. factors not significant enough, S-Check gave up looking for large delay value. 2. response times not linear with respect to delay value due to unusual program or varying system loads.	1. re-evaluate factor choices or set the delay manually. 2. look for unusual behavior in program or run when system load is stable.
resolution of delay value may be too coarse	1. S-Check's nominal delay value is too coarse for factor set.	1. response times may be unnecessarily long due the coarseness of S-Check's delay value. Future S-Check versions will allow for finer grain delay values.
runtime with no delay set is > runtime with delay	1. response interval does not encompass factors 2. selected factors are not significant enough to cause response time to increase significantly 3. program is very unusual	1. make sure the response interval encompasses all selected factors 2. re-evaluate factor set 3. check for severe communication contention or other anomalies which make the program run faster with delays. See [1] for an example.

TABLE 2. Experiment Control Warnings and Error Messages

Message	Reason	Fix/Consequent
variance computed with "*x*" degrees of freedom. You need at least 10 for a meaningful estimate.	1. plan does not provide enough information to obtain a meaningful estimate of the standard error	1. disregard and use normal probability plots to select significant factors. 2. choose new plan or run with replication.
S-Check can't handle saved suspended jobs. Job Status is set to built.	1. experiment was saved while job was suspended.	1. Current S-Check implementation can't handle suspended jobs. Job must be re-started.
Creation of instrumented source failed.	1. The program **CInstGen** failed.	1. Check the standard error message log and look for a work around. 2. This is a bug in S-Check and should be reported.
linking failed	1. unable to create executable	1. verify program code and linkage flags.

Table 2 gives a list of potential warning and error messages along with suggestions on how to resolve problems.

Setting the amount of delay

The delay value is either set by you manually or by S-Check automatically. By selecting *Build* (see *Running an Experiment*), S-Check will automatically determine an appropriate delay size for the experiment. It is part of the *build* process. Alternatively, you can set the delay value manually.

To set the delay value manually, enter a positive integer in the Delay text field on the Experiment Control Window. A message confirming the delay value setting is printed in the message area. To get back to the default state of letting S-Check automatically determine the delay value, choose *Unset*.

The duration of the artificial delay is an important aspect. Ideally, the delay should be long enough so that it can be distinguished from experiment noise and short enough so as not to produce unnecessarily long program execu-

tion times. When setting the delay manually, it is up to you to select an appropriate value. When the delay is determined automatically, a few trial runs are performed until a satisfactory setting is found.

The delay itself is a function that performs artificial instructions. The delay value controls how many times a loop iterates performing the artificial instructions. With the current implementation of the delay function, it is recommended that S-Check tests be performed without compiler optimization. A future improvement to S-Check will incorporate a variety of artificial perturbation options.

Selecting a DEX plan:

A design of experiment plan is a complete description of a minimum set of perturbation patterns needed to carry out a meaningful program investigation. A variety of schemes (briefly described below) are available in S-Check. It is important to note that there exists a direct correlation between the minimum number of runs in an experiment, the quantity of information provided by the investigation and the selected plan type. The trade off buys information with number of runs.

The user has six options for defining the experimental plan.

- Automatic
- Full factorial
- Half factorial
- Quarter Factorial
- Resolution IV
- Resolution III

The total number of factors under study is an important criterion in the choice of an experimental plan. For efficiency purposes, it is essential to maintain a balance between the quantity of information desired and the cost of the corresponding experiment. Any knowledge on interactions or the lack thereof should be put to use--there is no point looking for third-order interactions when they are known to be absent. Table 3 is intended to help you meet these requirements.

TABLE 3. Plan Selection

	Plan Selection		
	Up to S- Check	Up to the user	
Number of factors	<200	small(<12)	large(12<f<200)
Option	Automatic	Full Half Quarter	Resolution IV Resolution III

With *Automatic*, S-Check decides which plan type to use in investigating the program. Table 4 summarizes how choices are made.

TABLE 4. Automatic Selection

Number of Factors	Plan Type
up to 4	Full Factorial
5 or 6	Half Factorial
7 or 8	Quarter Factorial
9 to 31	Resolution IV
31 to 199	Resolution III

Full factorial designs consider every combination of factor levels, that is, they support inferences about all factor interactions. Be aware that the number of measurements (program runs) rapidly becomes prohibitive as the number of factors increases. A full factorial design is usually not appropriate for large numbers of factors. Note that high-order interactions are quite often of negligible magnitude when compared to main effects and low-order interactions; therefore when the number of factors increases, the desired information can be obtained by performing only a fraction of a full factorial experiment. Full factorial designs can currently be requested for up to 12 factors.

A *half factorial* design requires only half the runs of a complete full factorial. A minimum of 3 factors (theoretical limit) is necessary to request this type of plan. Such a design in 3 factors is of resolution III (explanation fol-

lows). To get an estimate of the standard deviation (called standard error) under this particular configuration you will need replicates. Other half factorial designs provide an estimate of the standard error without replication. Half factorial designs can be requested up to 9 factors (implementation limit).

A *quarter factorial* design requires only one-fourth the runs of full factorial (i.e., half the runs needed in a half-factorial design). A minimum of 5 factors (theoretical limit) is necessary to request such a design. This 5-factor configuration is of resolution III and as noted previously replicates are needed to get an estimate of the standard error. Other quarter factorial designs provide an estimate of the standard error without replication. This type of plan can be requested up to 10 factors (implementation limit).

The resolution of a plan gives an immediate indication of the information capacity of the design. *Resolution III* plans confound (mix) results of first and second order effects, while *resolution IV* plans do not confuse first and second order effects, but instead confound first and third order effects. Ideally one should only use plans of resolution IV and up. Both resolution III and resolution IV plans are, however, appropriate for preliminary screening of a large number of factors, since in these cases many factors are insignificant.

Resolution IV designs may be requested for any number of factors between 2 and 199. They provide information on main (first order) effects only. In contrast to resolution III designs, no replication is needed to get an estimate of the standard error.

Investigations built on resolution III plans require half the runs of those built on resolution IV plans. Again, information is provided only on main effects and this time replications are needed to get an estimate of the standard error. Resolution III plans may be requested for any number of factors between 2 and 199.

Setting Replication

Replication indicates the number of times the experiment is performed. The default value is 1 (minimum), which means the experiment is run once. To duplicate the experiment, set replication to 2. The range of the replication value is currently 1 to 99.

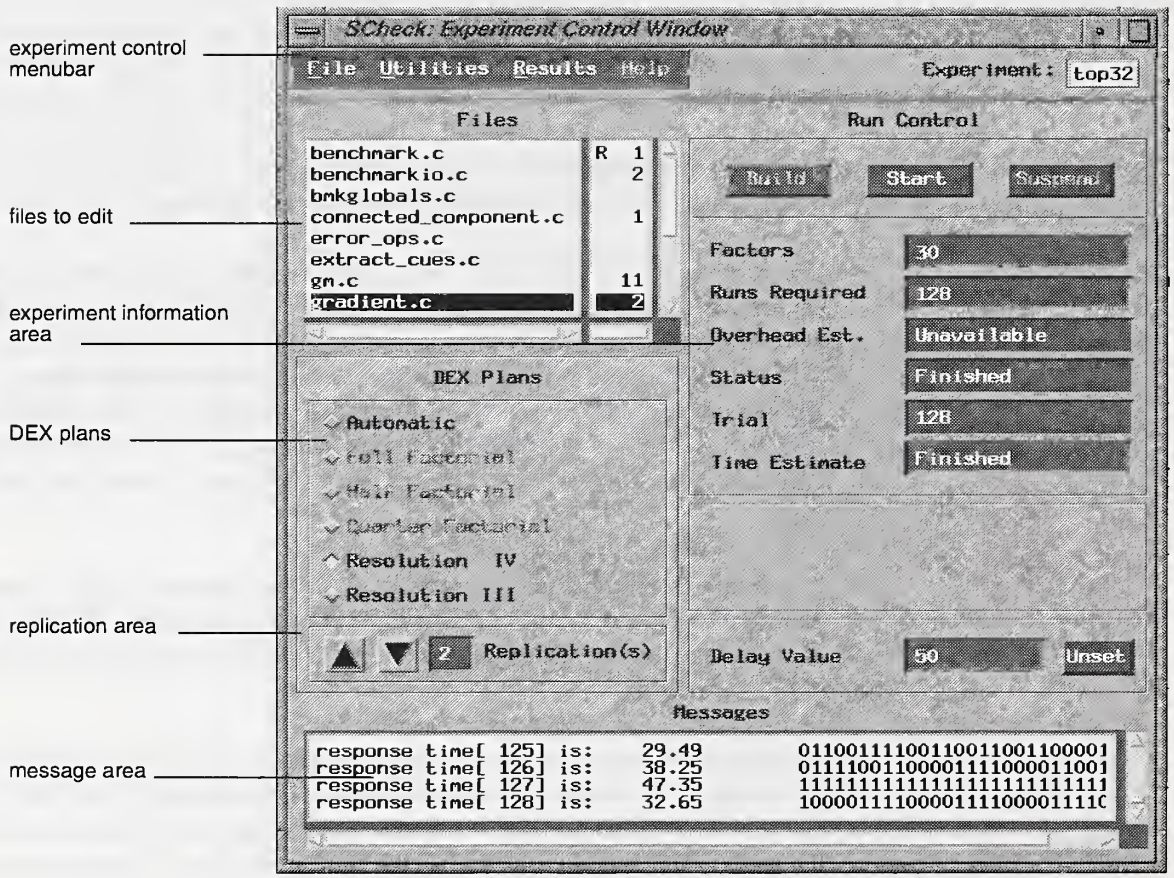


FIGURE 18. Experiment Control Window

Replication may be necessary if anticipated variability or noise level of the experiment is high. Shared-memory systems generally have lower experiment noise than distributed-memory machines, with "PVM" style experi-

ments being the worst. Replication may also be used to obtain a standard error for an experiment when a single run of the DEX plan cannot provide one. As mentioned, under some circumstances full factorial, half factorial, and quarter factorial plans do not provide a standard error. A single run of a resolution III experiment never provides a standard error (Normal probability plots allow some choices to be made without knowing the standard error). Refer to Appendix B for details on the availability and use of the standard error for an experiment.

To increase/decrease the number of replications, press the *up/down arrow* buttons to the left of the replication text field. For rapid movement hold the desired arrow button down. To reset replication to its default value click once in the replication text field.

Experiment Information Area

This area describes experiment parameters and attributes of the current experiment. Factors is the total number of factors selected for the experiment. Runs Required is the total number of times the test program is executed to complete the experiment. This value is affected by the number of factors, DEX plan, and the replication value. Overhead Est. (Overhead Estimate) gives an estimate (as a percentage) of how much longer the test program will run on average. Status indicates the state of the experiment: This topic is expanded in the next section. Trial indicates the current trial number S-Check is running (e.g., 17 of 32--the 17th run in a plan that requires 32 runs). Time Estimate shows how long the experiment will take to complete, based on preliminary measurements performed while determining the delay value and the number of runs in the experiment. This estimated value is updated during the execution of the experiment. If you set the delay value manually, then the Time Estimate is not given until after the first trial run.

Running an Experiment

There is a two step procedure for running an experiment. The first is an intermediate step called *build* (select *Build* on the Experiment Control Window). Building the experiment compiles the work set files and constructs an

executable test program. The built program is instrumented with code that can activate/deactivate perturbation code at each factor location. Building the experiment also determines the size of the delay value (if not set manually).

At this point the experiment is ready for execution. You can select (if you have not done so already) or change the DEX plan and number of replications without rebuilding the experiment. Changing the DEX plan or replication value can alter the number of runs required. **If the factor set is modified, the experiment must be rebuilt.**

To run an experiment, press *Start* on the Experiment Control Window. Starting an experiment will randomly execute each program version defined by the experiment parameters. The plan row and response time for a given trial is displayed in the message area. The series of 1's and 0's indicate whether or not a delay was executed for a given factor. If set to 1, the delay was executed. The code is not disturbed if set to 0. The factors are mapped to the plan row (left to right) and can be identified by correlating them with the id on the Effects List Window.

An experiment is suspended and restarted by selecting the *Suspend* and *Restart* buttons respectively. Suspending an experiment terminates the current running instance of the program. Data from completed trials are saved. Restarting an experiment first runs the trial that was terminated during the suspend operation and then continues the experiment. Response information is appended to data previously captured. Stop terminates the experiment with no saving of state. The job status is returned to **Built**, the experiment can be restarted. Start/Stop and Suspend/Restart occupy the same button region, therefore only one of the option pair is available at any given time.

Status indicates the current state of the experiment. An experiment can have the following states:

Empty	The experiment has not been built.
Building	The test program is being compiled with instrumentation code.

Determining Delay	Sample trial runs are executed for the purpose of finding a suitable delay value.
Built	The experiment was successfully built and the delay value is defined.
Running	The experiment is currently running.
Queued	The experiment is queued waiting for start-up. **This feature is currently unavailable.**
Suspended	The experiment is suspended. State information of previously run program instances are saved.
Terminated	The experiment has been Terminated. No experiment state information is saved.
Calculate Effects	All trials ran successfully. Results are being calculated.
Finished	The experiment has completed. Results of the experiment can be examined.

Viewing Internal Results

S-Check displays intermediate (raw) results of an experiment on the DEX Information Window (Figure 19). You can access this window under the Utilities menu on the Experiment Control Window. It displays the raw internal information S-Check uses to calculate effects. This information is not to be used directly as a performance measure, but rather to verify the soundness of the experiment and/or to conduct further analysis of the data.

There are a number of options available to display the data. DEX Plan displays the pattern of delays used in the experiment. A row in the plan corresponds to a trial run. A column corresponds to a factor and is linked to an index assigned during factor selection. The DEX Plan can be displayed before or after the experiment is run. The response time for each run can also be displayed by selecting the Response times option. All of this data can be viewed in either Run or Plan order. The plan can be packed (*i.e.*, no spaces) with the Pack plan option.

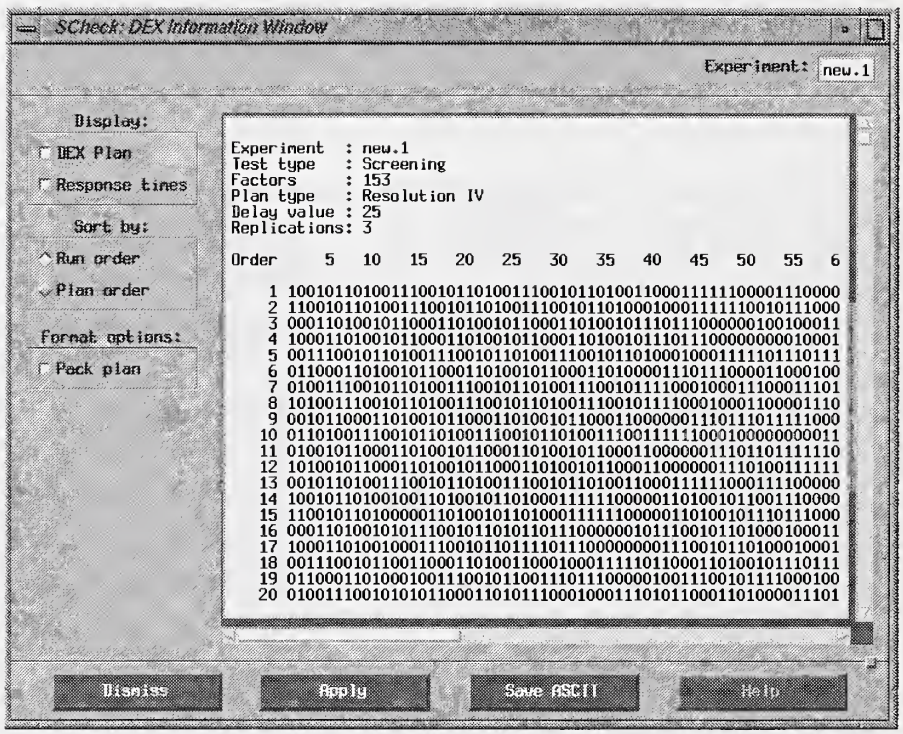


FIGURE 19. DEX Information Window

The data can be saved to a file in ASCII format and subsequently processed with statistical packages or programs outside of S-Check. One such situation may occur when a single Resolution III experiment is conducted without replication. Afterwards, it is deemed that a second run is needed to obtain a standard error and to verify results of the first run. Two options exist, the first is to rerun the experiment, setting the replication to 2. We would get the information desired, but at a higher cost since we had to ignore results of the first experiment and run it again with replication set to 2. The other alternative is to save the internal results of the first run and then make another run of the experiment (replication = 1). We could then save the internal data of this (the second) run and combine it with the internal data of the first run and calculate performance information from this combined file. However, these calculations would have to be performed outside

of S-Check with a statistical package or program. Such programs can be provided upon request.

Obtaining an External Profile

S-Check provides the capability to obtain a conventional external profile (*e.g.,* gprof or pixie) of your test program. The external (or UNIX) profile can help you select candidate test points and can be used to compare and contrast results from the profiler and S-Check. To obtain an external profile, select the *External Profile* button under the *Utilities* menu on the Experiment Control Window. Figure 20 shows the External Profile Window.

Select the profiler that you wish to execute. A default setting of parameters is given for Prof, Gprof, and Pixie. You can keep the default settings or set them for your particular system or preferences. The Generic selection provides the option to run other profilers not listed.

Make sure the appropriate library path is set (if necessary) for a given profiler. Pixie, for example, may require the LD_LIBRARY_PATH to be set to the current directory so that it can find libc.so.1.pixie. Also, by default prof will look for the count file exe_name.Counts. However, if your program creates multiple processes, pixie generates multiples count files with process id extensions. In this case the View command will need to be prof exe_name -pixie exe_name.Counts.*. Make sure that there aren't any *old* .Counts.* files around. These files reside in the "experiment_directory/.scheck/experiment_name" directory.

The Executable Name text field holds the name of the executable. The default, "exe_name" is just a place holder for the executable. It can be any name. The next two fields hold the compiler and linker flags. The Instr. field (for instrumentation) is provided if an instrumentation program is needed to patch code into the executable (*e.g.,* in pixie). The Run text field contains the command to execute the program and View contains the display command. Once all profile instructions are set, click-on Apply to execute the program and display results. Upon successful completion, results of the profile will appear in the Profile Data Window.

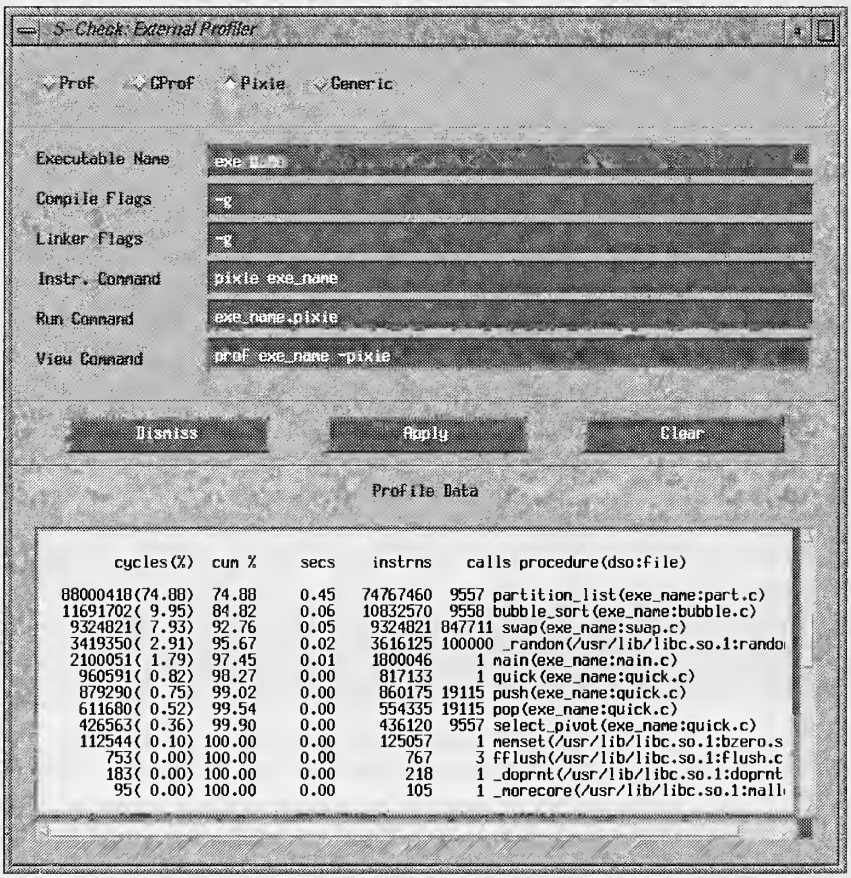


FIGURE 20. External Profiler

CHAPTER 6

Viewing Performance Data

There are two windows to view S-Check's performance data. You can either obtain a rank-ordered list of the effects by launching a List Effects Window or observe the effects in various plot views of the Plot Effects Window. By default, data from the current experiment are displayed when either of these two windows are launched. To view results of other experiments you can bring up the Multiple Display Window. These options are found under the *Results* menu on the Experiment Control Window.

List Effects

The List Effects Window (Figure 21) displays results of the experiment. At the top of the results area an experiment summary is given. The summary includes experiment settings and the standard error. After this the effects for each factor and factor interaction are displayed. The list follows this format:

index effect order term [line #]file function text

where:

index	arbitrary identification assigned to a factor or factor interaction
effect	expresses importance of corresponding factor or factor interaction
order	describes the degree of interaction. An order of 1 indicates a main (standalone) effect. An order of 2 indicates a 2-factor interaction and so on.
term	describes the factor or factor interaction via an index. Index and term are the same for main effects. For effects with an order of two or more, the term is represented by combining the indexes of constituent factors. The indexes are delineated by periods (.).
[line #]file	indicates the file and line number for the factor. If the factor is an interaction then that is indicated instead.
function	is the function in which the factor resides.
text	displays the source code corresponding to the factor.

The list control area gives the user the option to set preferences for displaying the effects. You can control a standard error filter, the level of the order to be displayed, and the sorting of effects by index or by value.

Setting the standard error filter determines how significant an effect should be to be displayed. For example, if the standard error is 0.25 and the standard error filter is set to 5, then only effects that are greater than 5 * 0.25 = 1.25 are displayed. Press *Apply* (or hit <return>) to activate the new setting. The default value for the standard error filter is one standard error (if activated).

The level of interaction, or order, of effects to be displayed is controlled by selecting the desired order in the order level area. Selecting the order will display effects of that order. More than one order may be selected. Once all

your selections are final, press *Apply* to see the changes. The default displays only main effects, but in parallel processing, second order effects cannot be dismissed.

Effects are sorted either by their value or by their index. Select the desired ordering and press the *Apply* button to activate the changes. Sort-by-value is the default.

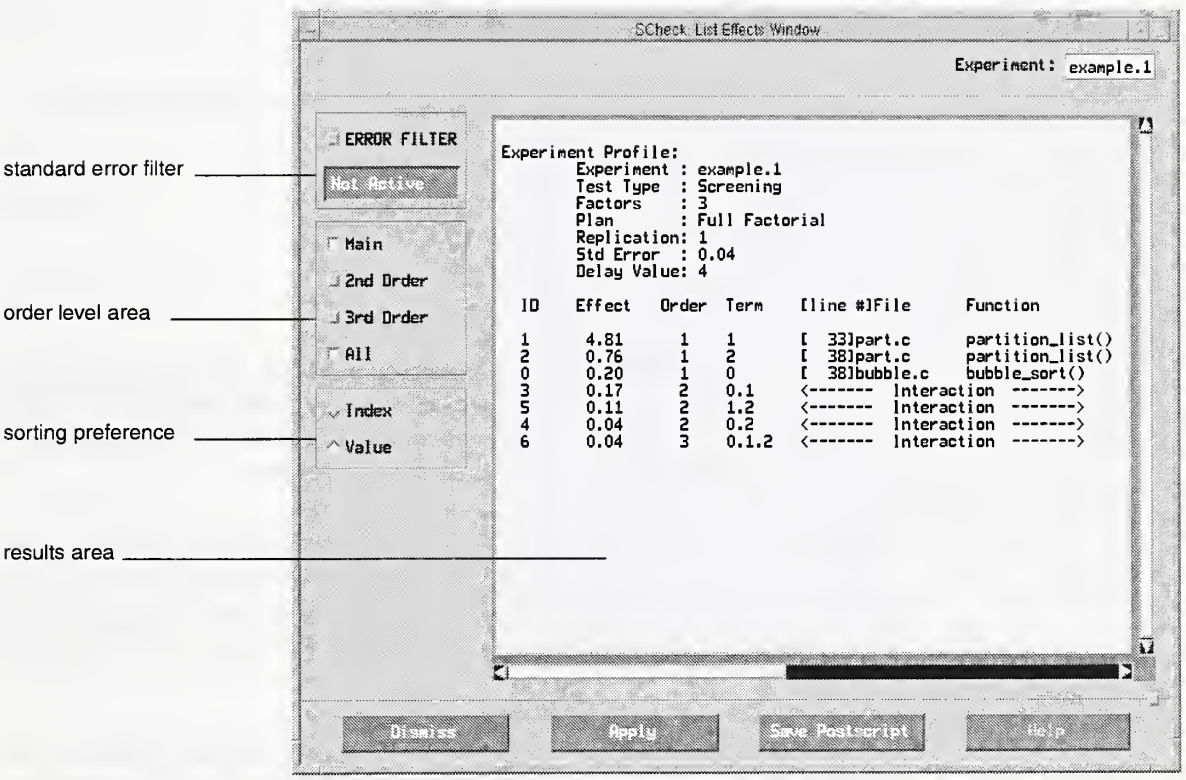


FIGURE 21. List Effects Window

If you are changing more than one setting in the list control area, you can wait until all of your selections are made before pressing *Apply*.

List information can be saved in a postscript file. Select the *Save Postscript* button. This action brings up a dialog containing a list of files in the current working directory and a text entry field. Enter the name of file you wish to save the effects list in and click the *Save* button. The implementation saves the currently selected display. All postscript files are saved in the *experiment directory*, regardless of the experiment.

Plot Effects

Four plots are available to visualize S-Check analysis results (Figure 22):

- absolute effects plot
- mean plot
- normal probability plot
- half normal probability plot

By default, all four plots appear in the display area when the Plot Effects Window is popped up. Each plot can be requested individually by clicking on the plot or by using one of the available push buttons. Clicking again in the display area takes you back to the previous display.

The plot control area gives you the option to set preferences for displaying the effects. You can control a standard error filter, the level of the order to be displayed, and the sorting of effects by index or by value. Unless otherwise specified, these options apply only to the absolute value of effects and half effects plots.

Next and *Previous* allow you to browse through the results whenever a single screen cannot handle the complete set of results at once. A screen can plot up to 32 data points.

The error filter applies only to the absolute value effects plot. It can be used to ignore factors that are not considered to be significant. To use this filter, select the *ERROR FILTER* toggle button. If a standard error exists, this will activate the text field immediately below the toggle button. Enter the stan-

dard error factor and press <return>. This action will highlight effects that surpass the threshold set by the filter. An effect must be greater than the standard error multiplied by the standard error filter to be highlighted (displayed in another color). In addition, a highlighted horizontal line is drawn at the threshold limit. The default value for the standard error is one (if activated).

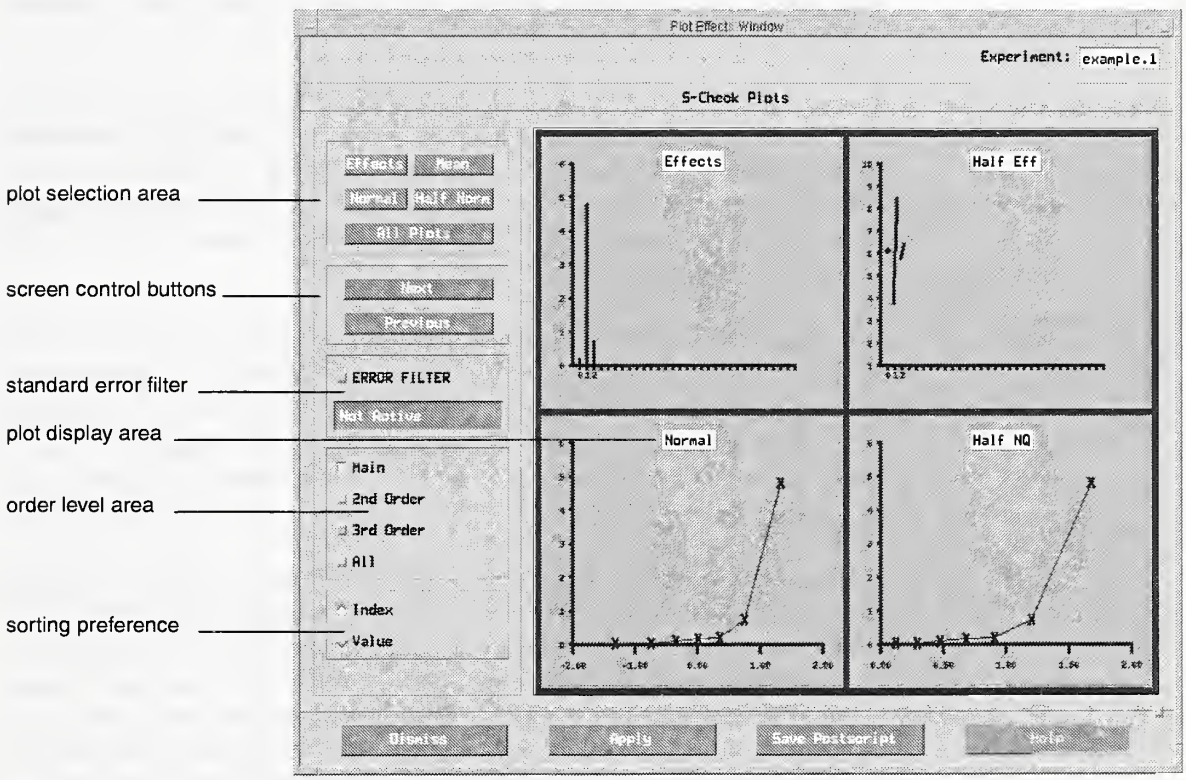


FIGURE 22. Plot Effects Window

The level of interaction, or order, of effects to be displayed is controlled by selecting the desired order in the order level area. More than one order may be selected. Once all your selections are final, press *Apply* to see the changes. The default displays only main effects.

Interaction availability is dependent on the experiment type. *Order* level toggle buttons are *grayed out* when interactions are not available.

Effects are sorted either by value or by index. Select the desired ordering and press the *Apply* button to activate the changes. In contrast to the List Panel, sort-by-index is the default.

If you are changing more than one setting in the plot control area, you can wait until all of your selections are made before pressing *Apply.* A description of each plot follows:

Absolute Effects Plot. The absolute effects plot represents the absolute values of effects for the different factors under study and their interactions (when available). The taller the line, the larger the absolute value of the effect (y axis). Numbers on the x axis are arbitrary identifications assigned to factors. Use the rank-ordered list (List Effects Window) to link indexes with their corresponding term.

Mean Plot. The effect of a factor is given by the difference in the means for all high and all low settings of the factor. The mean plot is generated by drawing a line between the mean at the low and high setting of a factor for every factor and factor interaction (when available). The longer the line, the larger the effect. As with the absolute effects plot, factors are identified by an arbitrary index (x axis).

Normal Probability Plot. In a normal probability plot, effects (y axis) are arranged in ascending order of their value (i.e., from smallest to largest) and plotted against the theoretical standard normal percentiles (x axis). Theoretical standard normal percentiles are the cumulative values one would expect if the effects arose from a normal distribution centered about zero with a standard deviation of one. If the effects in an experiment are generated purely by noise, then the points would tend to fall on a line passing through the origin. Outliers from this line indicate significant effects and so highlight potential performance bottlenecks. Data points (i.e., effects) are arranged in ascending order of their value.

Half Normal Probability Plot. The half normal probability plot is similar to the normal probability plot, except that effect estimates are arranged by

their absolute values (y axis) and plotted against the theoretical standard half-normal percentile (x axis).

The plots can be saved in a postscript file. Select the *Save Postscript* button. This action brings up a dialog containing a list of files in the current working directory and a text entry field. Enter the name of file you wish to save the plots in and click the *Save* button. The current implementation saves all four plots by default, there is no feature available yet to save individual plots. In addition to the plots a list correlating the factors indexes to their corresponding terms is provided. All postscript files are saved in the *experiment directory*, regardless of the experiment.

Saving Results

Results from the current experiment can be saved in a file by using the Save Results Window. The Save Results Window is accessed by selecting the *Save Results* button under the *Results* menubar. As mentioned earlier, it can also be accessed by selecting the *Save Results* toggle button from *Save* under the *File* menu. The Save Results Window can only be accessed when valid experimental results exist.

At the top of the Save Results Window are listed previously saved result files for the experiment. The text field (at the bottom of the window) provides an area to enter the name of the results file. Saved result files can be retrieved and viewed via List and Plot Windows through the Multiple Display Window.

Viewing Multiple Displays

The Multiple Displays Window (Figure 23) allows the user to view and compare results from previously saved experiments without having to load each experiment separately. List and Plot panels can be launched for any results file listed in the results list. This list displays all the result files saved for all the experiments under the current experiment directory. Results can

be saved by accessing the *Save* menu selection or the *Save Results* menu selection on the Experiment Control Window. If you want to display a results file, click in the corresponding toggle area and press List or Plot to obtain a display of the data. More than one results file may be selected at a given time. To see multiple results together, you must remember to move the displays around (they are stacked up initially) if your window manager automatically places windows.

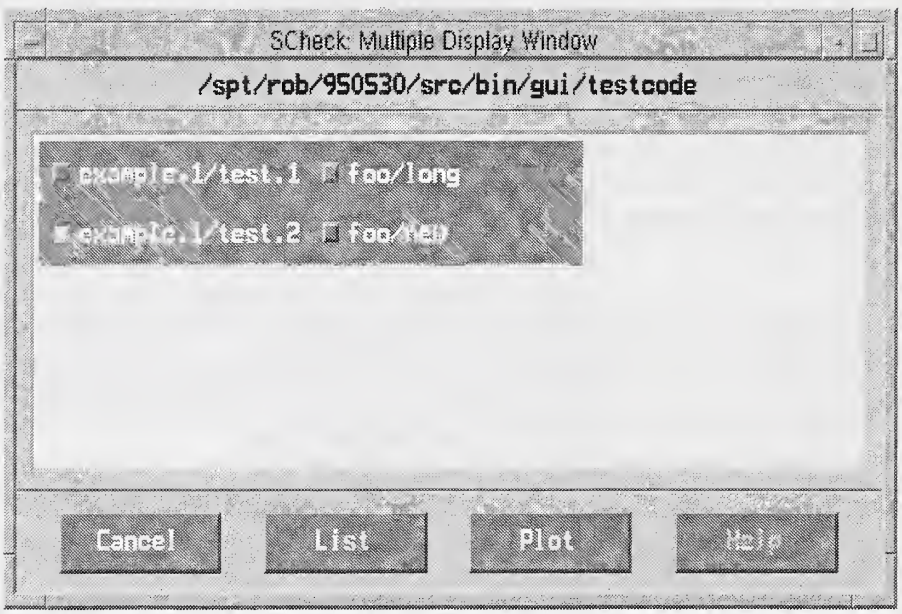


FIGURE 23. Multiple Displays Window

CHAPTER 7

Warnings and Bugs

Warnings

GCC Problems. Some later versions of GCC such as 2.7.2 have header files containing a lot of non-ANSI C keywords such as __const__, __CONSTANTVALUE, __attributes__(), etc. which cause S-Check's C language parser to fail. Most of these non-ANSI portions of the GNU header files can be shut-off by undefining the preprocessor symbol __GNUC__. This is done by adding the option -U__GNUC__ to the "Preprocesor" field of the Configuration Window. Note, as of this writing some header files distributed with GCC 2.7.2 did not utilize the __GNUC__ switch for the non-ANSI portions, so you must undefine them on an individual basis. See the note below "Use of non-ANSI types in the C language" for more details.

Irix 6.2 Problems. Under Irix 6.2, the MIPS and MIPSpro C compilers have header files containing the non-ANSI C type "long long," which under some conditions causes the S-Check C language parser to fail. For more details see the note below "Use of non-ANSI types in the C language."

Use of non-ANSI types in the C language. S-Check's C parser doesn't handle some commonly used non-ANSI types correctly such as "long long". Often it does not complain and the types are regenerated by the source code generator. However, under some infrequently occurring conditions this causes a corruption of the parse tree, and subsequent corruption of the instrumented code (the code will be missing some symbols) which causes the build to fail. In some cases, a work around maybe available. For example, in the "long long" case, use "long" if possible (i.e., does the code really need 64-bit integers).

NULL (;) statements in C are handled incorrectly. In some cases, the presence of a NULL (;) statement may cause incorrect behavior of the factor editor. Work around: remove the NULL statements.

C preprocessor anomaly. Version 3.0 of S-Check takes advantage of the target machine's C processor by using the '-E' option to cc(1). Therefore, S-Check only parses the preprocessor output, not the original code. This can cause the following problem, consider:

```
foo.h:
    #ifndef CORRECT
    you forgot to define CORRECT
    #endif
foo.c:
    #define CORRECT 1
    #include <foo.h>
    main(){}
```

Since S-Check never parses the macro CORRECT, the attempt to build an instrumented executable will fail. These problems will be corrected when a C preprocessor is integrated into S-Check. A work around is to add the #define to the special C-flags (i.e., -DCORRECT) for the file(s) in question on the Configuration Window.

Source code modifications and instrumentation. Modifying the source code will invalidate all instrumentation (*i.e.,* selected factors and/or the response interval) in the modified code. The user is notified of this upon recalling an affected experiment.

Cannot allocate colormap entry for "color". The server can't allocate the colormap entry for "color". Too many X-clients are currently running on the system. Exit one or more of these clients if you want to display the default S-Check colors.

Bugs

Running experiments simultaneously. Results from running two or more experiments simultaneously are unpredictable. This feature has not been thoroughly tested and should be avoided. In addition, running experiments simultaneously where the test program uses shared files (e.g., on SGI machines, shared arena) may cause problems.

Instrumenting switch statements. As mentioned in *Selecting Factors*, caution should be exercised when defining factors in compound statements. This is especially true when instrumenting switch statements. Selecting the switch statement itself will place the instrumentation code at the bottom of switch (by design), which under most circumstances is never reached. It is best to place the factors in the individual cases that you want to instrument.

Instrumenting any part of the Response Interval in a switch statement should be done with extreme care. S-Check is unable to detect the bounds of a case because it is not a compound statement, but rather a label.

Changing the language type of a fully built experiment. If you change the language type of a fully built experiment, S-Check eventually fails.

GLOSSARY

Glossary

analysis of variance (ANOVA)	statistical method used for the purpose of calculating effects.
effect	expresses importance of a factor or factor interaction.
experiment	an S-Check entity that defines parameters needed to perform an SPS analysis of the test program.
experiment configuration	items that define settings that allow S-Check to build (make) the test program. It also defines other setup information such as where to run the experiments and the type of the experiment.
experiment control settings	items that define settings directly associated with the experiment, e.g., factors, plan type, delay value, and replication.
experiment directory	the directory in which S-Check was started. Experiments are saved in this directory under the sub-directory named *.scheck*.
delay value	the amount of delay.
DEX	design of experiments.

factor — parameter that is being varied and tested (e.g., source code segments, synchronization barriers, etc.).

factor profile — a list which provides the number of certain constructs (e.g, a for loop) in a particular file or files. The profile can be used to aid factor selection.

host machine — the machine on which S-Check is running.

platform — the environment in which S-Check experiments will run.

replication — indicates the number of times an experiment is performed.

response interval — defines the start and stop locations for the purpose of capturing the response time.

response time — actual execution time for response interval. A response time is captured for each trial run.

SPS rank — rank-ordered list of source code segments based on the relative sensitivity (effects) of the test program to synthetic delays associated with the code segments.

standard error — estimate of the standard deviation of observed effects.

target machine — the machine on which the S-Check experiments are performed.

test program — executable test program for the experiment.

treatment — instrumentation pattern (of delays) obtained from the experimental plan for a trial run.

trial — is an instance of the test program with a particular treatment.

work set — set of source code files needed to build the test program.

working directory — the directory in which files are accessed to build the test program. This directory will either be the directory in which S-Check was started (experiment will run on local machine) or the directory specified on the Configuration Window for remote machine access.

References

[1] G. Lyon, R. Snelick, R. Kacker.
Synthetic-perturbation tuning of MIMD programs,
The Journal of Supercomputing 8 (1) (1994), 5-28.

[2] R. Snelick, J. JaJa, R. Kacker and G. Lyon.
Synthetic-perturbation techniques for screening shared memory programs,
SOFTWARE--Practice and Experience 24 (8) (1994), 679-701.

[4] R. Snelick.
S-Check: A Tool for Tuning Parallel Programs,
Proceedings of the 11th International Parallel Processing Symposium,
Geneva, Switzerland (April 1-5, 1997), 107-112.

[4] R.Snelick, M. Indovina, M. Courson, A. Kearsley
Tuning Parallel and Networked Programs with S-Check,
Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97),
Las Vegas, Nevada (June 30 - July 3, 1997) Volume I, 21-30.

[5] R. Snelick.
S-Check, by Example.
NISTIR 6022, (available at **ftp:cmr.ncsl.nist.gov**)

[6] G. Lyon, R.Kacker, and A. Linz.
A scalability test for parallel code,
SOFTWARE--Practice and Experience 25(12)(1195) 1299-1314.

[7] IBM RS 6000/SP Parallel Environment: Operation and Use,
Volume 1.

APPENDIX A

Error Messages

Appendix A gives a list of common error messages from S-Check. Errors of this sort are the result of invalid user input or input that is unrecognizable by S-Check. Adjustments (or input corrections) must be made before S-Check can proceed in the direction before the error occurred. Note that all other system functions are frozen until the user acknowledges the error by clicking on the OK button in the error popup dialog. The table below provides possible reasons and solutions to the errors. If you are unfamiliar with how S-Check instruments the source code, it may be useful to review Appendix C (Code Instrumentation). The information provided there will let you better understand the error messages.

Message	Reason/Fix
The program "X" is not in your path.	The executable "X" is not contained in any of the directories specified by your PATH variable. You must quit S-Check and correctly modify your PATH and restart S-Check.
The program "X" received a "Y" signal.	The program "X" received a termination signal "Y" from the UNIX operating system. If "X" was **CInstGen** or **Cparser**, this is an S-Check bug and should be reported. If "X" was **a.out**, a bug in the user's test program exists. If "X" is **cc**, this is a compiler bug.
The program "X" received unknown signal "Y".	The program "X" has received an unknown signal "Y" from the Unix operating system. "X" can be one of the following: **a.out**, **cc**, **Cparser**, or **CInstgen**.
"X" is not a regular file	S-Check expected a regular file. Check file type of file "X".
The generator has returned unknown code "X".	The program **CInstGen** has terminated with an exit status unknown to S-Check. This is a bug in S-Check and should be reported.
Invalid delay value: "X", delay is not set. OR Invalid delay value: "X", delay is still "Y"	The delay value entered manually does not fall in the range 0 to DELAY_MAX_VALUE. DELAY_MAX_VALUE can be modified when S-Check is installed. It's default value is 1,000,000.
CTree_Load failed	Unable to load the "X.tree" file because it does not conform to a known format.
Ran out of memory for source file load.	Ran out of memory (malloc() failed). Close other windows, stop other processes, increase swap space, add memory, etc.
bad read for file "X"	File "X" does not conform to a known format. If "X" is config then experiment save file is corrupted. You may not be able to salvage the saved experiment.

Message	Reason/Fix
bad read for response.scheck file, execution time unknown	The file response.scheck does not conform to a known format. The Begin (B) and End (E) markers were probably improperly placed. Check to see if the E was executed, or that E was executed before B.
bad factor kind found in config file	The config file does not conform to a known format. The file may be corrupted or the file may be incompatible with the current S-Check version.
illegal stmt index for factor in config file	The identified factor index is out of range. The config file is corrupted. The factor in question is removed from the factor list.
The parser has returned unknown code "X"	This is an S-Check bug and should be reported.
Invalid file name, file not saved.	An invalid file name was entered. The file was not saved, try another name.
Can't load results file: "X"	The results file "X" can not be read, contains corrupted data, or host system memory is low.

APPENDIX B

Standard Error Table

The standard error of an experiment is used to assess the quality of S-Check's results (rank lists and plots). However, depending on the experiment, a standard error may not be provided or its accuracy may be limited. In these cases, valid conclusions can still be made from S-Check's results, although with diminishing confidence.

The table below shows availability of the standard error for an unreplicated experiment. The standard error for unreplicated plans is calculated by treating interaction effects of three or more as error. When third order interaction effects are not available, second order interaction effects are used for the standard error estimate. This is the case for resolution IV plans when the number of factors is ten or greater.

The plan and the number of factors determine the accuracy of the standard error. A "YES" indicates that the standard error is calculated with sufficient precision and can be used with confidence in analyzing the results. A "YES*" indicates that a standard error is provided, but it may not be very reliable. Typically, unreplicated plans are used when the number of factors is six or more. When the number of factors is less than six, it is generally possible to replicate the experiment and calculate the standard error from replication.

A "NO" indicates that no standard error is calculated for the experiment. In the "YES*" and "NO" cases, inferences of the data can still be made but it should be done judiciously. For example, when the system load (and noise) is low, the estimate of the standard error may not be necessary. In such cases a resolution III plan may be used for screening. In contrast, investigating interactions between send/receive pairs in a highly congested (noisy) communication system may warrant a sound standard error estimate.

	Factors						
Plan	**2**	**3**	**4**	**5**	**6-7**	**8-9**	**10+**
Full Factorial	NO	YES*	YES*	YES	YES	YES	YES
Half Factorial	N/A	NO	NO	NO	YES	YES	YES
Quarter Factorial	N/A	N/A	N/A	NO	YES*	YES	YES
Resolution IV	NO	NO	NO	NO	NO	NO	YES
Resolution III	NO	NO	NO	NO	NO	NO	NO

NO: no standard error is possible for the plan

YES: the standard error is calculated with sufficient precision

YES*: the standard error is calculated with insufficient precision

N/A: the plan is not available for the specified number of factors

Note: a standard error is always available when the experiment is replicated.

Decisions about the importance of the standard error for analyzing S-Check's results can be aided with information about the test program and host system. Knowledge from previously-run experiments is especially useful. In addition, the significance of a factor in relationship to other factors in lieu of adequate standard error information can still be determined through the use of normal probability plots. If significant outliers exist in the results, these plots highlight them as points not on a straight line.

APPENDIX C

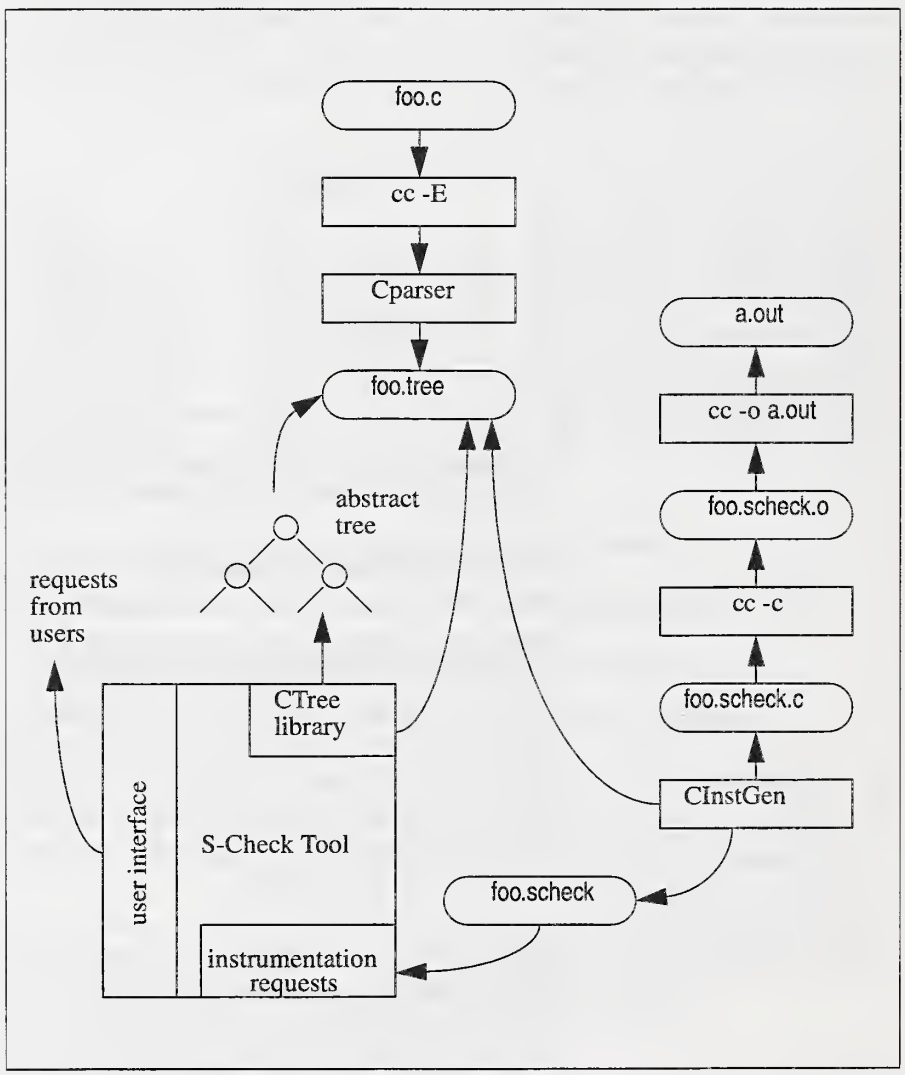
Code Instrumentation for the C Language

Appendix C explains the method in which S-Check instruments the source code program.

Given the file foo.c, S-Check runs the default C preprocessor for the purpose of including include files and expanding macros. The source code output from the C preprocessor is then analyzed with the **Cparser** program. The **Cparser** builds an abstract tree containing lexical and semantic information about the source code. This information is stored in the file foo.tree. A separate tree is built for each source file. S-Check (CTree library) uses the abstract tree to make inferences about the code and to re-generate the original source code with instrumentation. Inferences about the code allow for basic block factor selection, requests such as "instrument all while loops in function X()" are possible.

To instrument and re-generate the source code, two inputs to the code generator program (**CInstGen**) are required: the abstract tree and the user's instrumentation requests. Instrumentation requests are gathered and translated by S-Check and are saved in foo.scheck. The files foo.scheck and foo.tree are fed as inputs to **CInstGen** which yields the instrumented source code (foo.scheck.c). This code is then compiled with a compiler of the user's

choice to obtain object code (foo.scheck.o). Other object files are then linked together to create the executable program.



APPENDIX D

Linking S-Check to Batch Queuing Systems

Appendix D describes how to build Unix shell scripts to interface S-Check to various batch queuing systems (BQS). The advent of distributed computing has sparked a slew of systems where a scheduler program controls jobs on the computing system. Examples of such scheduling systems include LoadLeveler, DQS, and LSF. S-Check provides an interface to LoadLeveler and will provide interfaces to other systems in the future. However, the task of supporting all systems would be exhausting. In lieu of this, S-Check provides a platform type called Generic so that any queuing system can interface to S-Check provided that suitable scripts can be developed. We describe and show an example of how to write such scripts in this Appendix. The scripts need to be written and installed before S-Check can be started. Once this is complete, the configuration process proceeds as shown in Chapter 3 for the LoadLeveler example.

Writing BQS interface scripts. In order to be as generic as possible, S-Check calls a set of shell scripts to manage jobs in a queuing system environment. These shell scripts may be easily modified to suit an unusual configuration or a non-supported architecture. Each script performs one of the three basic actions:

- run the job

- kill the job
- check the status of the job

The requirements for each is described below:

1) Run (run.sh)

Runs the executable using system job control commands and returns the *id* of job for use by other scripts. It performs any necessary modification of job control scripts in order to set the *executable*, *pathname*, and *environment* variables.

Inputs:

executable-name - full pathname to the executable

trial_number - index of the current trial

command_file - name of the script file relative to the experiment directory

arguments - for the application as given in the configuration window

Output:

Job identification string, *e.g.,* "danube.1164"

Exit codes:

0 on success
1 on failure

Notes:

- the trial number starts at 1. This may be used for initializing the environment.
- for the last run, the trial number is prefixed with "_" (*e.g.,* _32). This may be used to trigger cleanup code.

- in addition to the command line arguments, **run.sh** receives some information from the following environment variables:

  SCHECKDELAY : delay value

  SCHECKFACTORS : factor array

The script should make sure that these variables are made available in the executable environment (for example using *export* for *sh*).

- the command file is relative to the experiment directory, which is located two levels below the directory holding the executable (*e.g.,* if the executable is *{some_dir}/a.out*, the command file is in *{some_dir}/../..).*

2) Kill (kill.sh)

Terminates a running job using the system's job control commands.

Input:
Job identification string, as returned by **run.sh**

Output:
none

Exit codes:
0 on success
1 on failure

3) Status (status.sh)

Checks run status of job using the system's job control commands and identifies it as one of three status indicators: pending, running, or complete.

Input:
Job identification string, as returned by **run.sh**

Output:
"PENDING"
"RUNNING"
"DONE"

Exit codes:

1 - job pending
2 - job running
3 - job complete
4 - indeterminate status

Example for LoadLeveler. Below are example shell scripts used to interface S-Check to LoadLeveler's queuing system. This example can be used as a prototype for writing interface scripts to link to your particular environment.

LoadLeveler Run (run.sh)

```
#!/bin/sh

# run.sh for IBM SP2 using LoadLeveler
# Input: executable trial_number command_file [args for the application]
#        trial_number prefixed with _ (eg. _42) if this is the last run
# Output: job identification string, eg "danube.1164"
# Note: when a copy is made from the original command file, the following
#      tags are replaced by their corresponding value

# @SCHECK_initialdir@        : initial directory where the job should start
# @SCHECK_program@       : local program name in the initial directory
# @SCHECK_arguments@   : user arguments as given in the configuration window
# @SCHECK_trialnumber@  : trial number
# @SCHECK_lasttrial@       : set to 1 if this is the last trial, 0 otherwise
# @SCHECK_delay@           : delay value as in the SCHECKDELAY env. variable
# @SCHECK_factors@       : factor array as in the SCHECKFACTORS env. variable
# @SCHECK_variables@     : insert some sh/bash/ksh code to set and export the
#                                SCheck environment variables in the command file

default_method=0
initialdir=`dirname $1`
program=`basename $1`
fullpath=$1
command_file=$3
# this holds the index of this trial
if [ `echo $2 | grep -c '^_'` -eq 0 ]; then
  trialnumber=$2
  lasttrial=0
```

```
else
  trialnumber=`echo $2 | sed 's/^_//'`
  lasttrial=1
fi

# get rid of the executable name, command_file and trial # to get the arguments
shift
shift
shift
arguments="$*"
# make sure that the critical tags are here, otherwise we'll use the
# default (stupid) method
if [ `grep -c "@SCHECK_variables@" $initialdir/../../$command_file` -eq 0 ]; then
  default_method=1
fi
if [ `grep -c "@SCHECK_program@" $initialdir/../../$command_file` -eq 0 ]; then
  default_method=1
fi
# make a copy of the command file, replacing the tags by their value
# Using a caret(^) as a field separator to be on a safer side
sed "
s^@SCHECK_initialdir@^$initialdir^g
s^@SCHECK_program@^$program^g
s^@SCHECK_arguments@^$arguments^g
s^@SCHECK_trialnumber@^$trialnumber^g
s^@SCHECK_lasttrial@^$lasttrial^g
s^@SCHECK_delay@^$SCHECKFACTORS^g
s^@SCHECK_factors@^$SCHECKDELAY^g
/@SCHECK_variables@/c\\
## S-Check variables added below ## \\
SCHECKDELAY=$SCHECKDELAY \\
export SCHECKDELAY \\
SCHECKFACTORS=$SCHECKFACTORS \\
export SCHECKFACTORS" \
   $initialdir/../../$command_file > $initialdir/scheck.$command_file

outfile=$initialdir/scheck.$command_file

if [ $default_method -eq 1 ]; then

  # issue a warning, but only once, for the first run
```

```
  if [ $trialnumber -eq 1 ]; then
    echo \
      "Warning: @SCHECK_program@/@SCHECK_variables@ not in $command_file" >& 2
    echo "Using the default method" >& 2
  fi
  # output the Scheck stuff at the end
  echo "## S-Check added the lines below ##"                    >> $outfile
  echo SCHECKDELAY=$SCHECKDELAY                                 >> $outfile
  echo export SCHECKDELAY                                       >> $outfile
  echo SCHECKFACTORS=$SCHECKFACTORS                             >> $outfile
  echo export SCHECKFACTORS                                     >> $outfile
  echo echo RUNNING SCHECK TEST with the default method         >> $outfile
  echo /usr/bin/poe $fullpath                                   >> $outfile

fi
if text=`llsubmit $outfile`; then
  # llsubmit returned 0
  echo `echo $text | cut -d' ' -f4 | sed 's/\"//g' `
  exit 0
else
  # llsubmit failed
  echo "ERROR"
  exit 1
fi
```

LoadLeveler Kill (kill.sh)

```
#!/bin/sh

# kill.sh for IBM SP2 running LoadLeveler
# Input  : job identifier
# Output : exit status (0 on success, 1 on failure)

if llcancel $1; then
  exit 0
else
  exit 1
fi
```

LoadLeveler Status (status.sh)

```
#!/bin/sh

# S-Check
# status.sh for IBM SP2 running LoadLeveler
# Input : job id
# Output: RUNNING | PENDING | DONE

retvalue=`llq $1 | awk "/^$1/  { printf \\$5 }" `
case $retvalue in
 I | P | D | S | H | SH | ST )
  echo PENDING
  exit 1;;

 R )
  echo RUNNING
  exit 2;;

 "" | C | RM | RP )
  echo DONE
  exit 3;;

 *)
  echo "Unknown status!"
  exit 4;;

esac
```

Index

A
Arguments text field 29
artificial delay 49
Automatic Factor Selection 17, 38

B
barrier test 6, 30
Batch Queuing Systems 28, 83
build. See experiment

C
C 3, 33, 81
C-flag 29
CInstGen 81
code instrumentation 36
Command line arguments 29
Configuration Window 15–31, 46
Configure menu selection 46
Cparser 81
CTree 81

D
default C-flags 29
delay value 42
delay value, setting the 48, 54
design of experiment (DEX) 2, 35, 49
Directory Contents 29
DQS 83

E
error
  filter 60, 63
error messages 48
Errors 75
exiting S-Check 12, 46
experiment 9
  building 48, 53
  creating 12
  deleting 12
  noise 48, 52
  opening 46
  plan 49
  restarting 54

running 53–55
state of 54
stopping 54
suspending 54
Experiment Control Window 12, 45–53
experiment design/plan
full factorial 49, 50
half factorial 49, 50
quarter factorial 49, 51
resolution III 49, 51
resolution IV 49, 51
experiment directory 11
experiment information
Factors 53
Overhead Est. 53
Runs Required 53
Status 53, 54
Time Estimate 53
Trial 53
Experiment List Window 12–15
External Profile 57

F

factor
selection 33–38, 46
Factor Editor 33, 35–42, 46
Factor Management 40
Factor Profile 38
factor selection mode 41
File menu 15
FORTRAN 3, 28, 33, 43

H

Help 3

I

IBM SP2 30
input/output redirection 30
interactions 49

L

LAM 17, 26–28
ld Flags 29
loader/linker flags 29
LoadLeveler 17, 18–23, 28, 83
LSF 83

M

messages, on Experiment Control Window 46
MPI 2, 17, 26–28

P

plots
Absolute Effects 64
Absolute Normal Quantile 64
Half Effects 64
Normal Quantile 64
POE 17, 23
postscript
save list effects 62
save plots effects 65
PVM 2, 17, 24–26

Q

Quit S-Check menu selection 46

R

replication, setting 52
response interval
setting 40–42
response time 40, 54
results
list effects 59–62
Multiple Displays 65
plot effects 62–65
saving 46, 65–??
viewing 59–66

S

Save As menu selection 12, 46
Save menu selection 46
scaling test 6, 30
SCHECKDELAY 85
SCHECKFACTORS 85
screening test 5, 30
Select Response button 35, 41
select response mode 41
Set Delay menu selection 48
SGI Challenge 15

Show Delay menu selection 49
SPS 1
SPS technique 5–6
standard error 79
Synthetic Perturbation Screening 1, 5

T
test program 11

V
Viewing Internal Results 55

W
warning messages 48
work set 29, 46, 53

How to install S-Check

S-Check software can be obtained via the ftp site **www.scheck.nist.gov**. You can also get to this location by accessing **http://www.scheck.nist.gov**. S-Check 3.0 release works on parallel SGI systems, IBM's SP machines, homogenous SUN, SGI, and RS6000 workstation clusters using PVM or MPI, and PCs running Linux. S-Check supports multiple languages including C and FORTRAN. The graphical interface is written with the OSF Motif toolkit (version 1.2). You must have the Motif libraries to compile S-Check.

Steps:

1. Retrieve the compressed tar file *scheck_3.0.tar.gz (or .Z)* from the scheck directory.
2. Use gunzip to un-compress the file.
   % **gunzip** *scheck_3.0.tar.gz*
3. Use tar to extract the files.
   % **tar** xf *scheck_3.0.tar*
4. Change directory to scheck_3.0
   % **cd** scheck_3.0
5. Configure the package for your system.
   % **configure**

   > This command automatically configures the software source code package for your particular system. Among other things, it builds the Makefiles necessary to create the executables.

6. Make the executables.
   % **make**
7. Make sure that your path includes the scheck_3.0/bin directory or move the executables (*e.g.*, **CInstGen**, **Cparser**, **CppFilter**, **scheck, etc.**) in this directory to a directory in your path.
8. Make sure the resource file **Scheck** is in your home directory on the machine your running S-Check on. It should be there if Step 6 executed correctly.
9. Refer to the user's guide, *"Using S-Check"* or *"S-Check, by Example"*, to get started with the tool.

Note: S-Check has a X-window user interface, so remember to set your DISPLAY environment variable to your local X-server and add the client machine to your xhost list. Use the command setenv DISPLAY your_X-server_machine:0 on the client machine to set the DISPLAY variable, and the command xhost +client_machine on your local X-server to allow the client machine to open a window on your console.

Send questions or comments to **scheck-tool@www.scheck.nist.gov**.

getting started

- put all ".c" parts of test program in same directory
- type the command scheck

S-Check Quick-Reference
Release 3.0
(some transitions omitted)
Written by: Gordon Lyon

SCheck: Experiment List Window

- select <name> from Experiment List Window to rerun or modify old experiment

or

- type in <new experiment name> to build completely new experiment

SCheck: Configuration Window

- select directory contents and add to work set. Set flags as needed for host parallel system. Exit via *OK* button.

SCheck: Experiment Control Window

- set *DEX Plans* on *automatic*, *Replication(s)* = 1 or 2
- double-click on files to establish or change response timing interval and code segments (factors) of interest

SCheck: Factor Editor Window

- click on code parts suspected as bottlenecks. A **black mark** will indicate selection. A second click deletes the selection. The window indicates the number of selected factors. Factor selection is always active unless you hit the response selection button. Exit via *OK*.
- the *Select Response* button in this window sets points *B-E* between which experiment timings will occur. This response is usually set to catch run time for the whole program, so the window margin markers of *B* and *E* should be set in the main driver. (The code being instrumented is indicated in the upper left-hand corner of the window.) The cursor will change to **red** and be directional during response point setting.
- exit of the window leads back to the Experiment Control Window, where other files can be set up or the experiment begun (see below).

setup continues

setup stage done

SCheck: Experiment Control Window

- hit the *Build* button to initiate compilation and selection of delay size. Or, you can use the *Delay Value* text field to set the delay manually, should you wish to do so.
- when the *Start* button becomes enabled, hit it to begin (errors will prevent this)
- when the experiment is done, use the *Display* menu selection to plot or list results.

List or Plot Effects Window

- if there is an error estimate, the error filter can be set in multiples of the standard error (*e.g.*, 2SE is a 95% confidence acceptance level). Any effect less than the Standard Error (SE) is probably not significant. Even 2SE is marginal for SCheck Technology.